DL_MESO USER MANUAL

M. A. Seaton and W. Smith

STFC Daresbury Laboratory
Daresbury, Warrington, Cheshire, WA4 4AD
United Kingdom

Version 2.4, May 2011

Contents

Contents i

1 DL_MESO General Information 1
- 1.1 Description . . . 1
- 1.2 Functionality . . . 1
- 1.3 Requirements . . . 2
- 1.4 The DL_MESO Directory Structure . . . 2
- 1.5 Disclaimer . . . 3
- 1.6 Copyright . . . 3
- 1.7 Authors . . . 3
- 1.8 Suggestions and Bug Reports . . . 3

2 The DL_MESO GUI 5
- 2.1 Getting Started with the DL_MESO GUI . . . 5
- 2.2 Lattice Boltzmann and the DL_MESO GUI . . . 6
- 2.3 Dissipative Particle Dynamics and the DL_MESO GUI . . . 11
- 2.4 Compiling and running DL_MESO . . . 17
- 2.5 Notes . . . 18

I Lattice Boltzmann Equation (LBE) 19

3 The Lattice Boltzmann Equation: Basic Theory 21
- 3.1 Introduction . . . 21
- 3.2 Basic Definitions . . . 21
- 3.3 Derivation of Equilibrium . . . 22
- 3.4 Structural Relaxation and Macroscopic Equations . . . 23
- 3.5 Mesoscale Interaction . . . 24
- 3.6 Summary of Lattice Boltzmann Equation . . . 24

4 DL_MESO_LBE Basic Definition 25
- 4.1 Lattice models . . . 25
- 4.2 Data structure . . . 32
- 4.3 The Parameters and Their Functions . . . 36

5 DL_MESO_LBE Features 41
- 5.1 Collision and Propagation Algorithms . . . 41
- 5.2 Boundary conditions . . . 44
- 5.3 Mesoscale interactions . . . 46
- 5.4 Diffusion and heat transfer . . . 47
- 5.5 Compressible and incompressible fluids . . . 48

6 DL_MESO_LBE Package Reference 49
6.1 Overview . . . 49
6.2 DL_MESO_LBE Subroutines and Functions . . . 50
6.3 DL_MESO_LBE Data Files . . . 101

7 DL_MESO LBE Examples 105
7.1 2D_Pressure . . . 105
7.2 2D_Shear . . . 105
7.3 2D_CylinderFlow . . . 106
7.4 2D_KarmanVortex . . . 106
7.5 2D_LidCavity . . . 106
7.6 3D_PhaseSeparation . . . 107
7.7 3D_Shear . . . 108

II Dissipative Particle Dynamics (DPD) 109

8 Dissipative Particle Dynamics: Basic Theory 111
8.1 Introduction . . . 111
8.2 Outline of Method . . . 111
8.3 Equation of state and dynamic properties . . . 113
8.4 Derivation of Equilibrium . . . 113
8.5 Summary of Dissipative Particle Dynamics . . . 114

9 DL_MESO_DPD Basic Definition 115
9.1 Data structure . . . 115
9.2 The Parameters and Their Functions . . . 118

10 DL_MESO_DPD Features 123
10.1 Domain decomposition and linked-list cell calculations . . . 123
10.2 Thermostats and integration algorithms . . . 124
10.3 Barostats . . . 127
10.4 Particle-particle interactions . . . 129
10.5 Long-ranged Electrostatic (Coulombic) Potentials . . . 132
10.6 Bond interactions between particles . . . 134
10.7 Surface interactions . . . 137

11 DL_MESO_DPD Package Reference 139
11.1 Overview . . . 139
11.2 DL_MESO_DPD Subroutines and Functions . . . 140
11.3 DL_MESO_DPD Data Files . . . 172

12 DL_MESO DPD Examples 179
12.1 Mixture_Ser . . . 179
12.2 PhaseSeparation . . . 179
12.3 Aggregate . . . 180
12.4 Polyelectrolyte . . . 180
12.5 AmphiphileMesophases . . . 181
12.6 Mixture_Par . . . 181
12.7 VesicleFormation . . . 181

A Manual compiliation and running of DL_MESO 183
A.1 DL_MESO_LBE . . . 183

A.2 DL_MESO_DPD 184

B DL_MESO Utilities **187**
B.1 DL_MESO_LBE 187
B.2 DL_MESO_DPD 188

C DL_MESO Licence Agreement (Academic Purposes) **193**

Bibliography **201**

Acknowledgements

DL_MESO was developed under the auspices of the Engineering and Physical Sciences Research Council (EPSRC) for the EPSRC's Collaborative Computational Project for the Computer Simulation of Condensed Phases (CCP5).

The members of the CCP5 DL_MESO consortium were:

David M. Heyes, University of Surrey

Chris M. Care, Sheffield Hallam University

Peter V. Coveney, University College London

David Emerson, STFC Daresbury Laboratory

Rob English, North East Wales Institute

Andrea Ferrante, Unilever Port Sunlight

Ian Halliday, Sheffield Hallam University

John Harding, University of Sheffield

Sebastian Reich, Imperial College

Bill Smith, STFC Daresbury Laboratory

Patrick B. Warren, Unilever Port Sunlight

Julia Yeomans, Oxford University

Many other people have given advice and encouragement in the development of DL_MESO. We gratefully acknowledge the support of the following people: Maurice Leslie, Richard Wain, Alexandre Dupuis, Jonathan Chin, Michael Dupin, Weiming Liu and John Purton.

Particular thanks go to Rongshan Qin at Imperial College as the original author of the Lattice Boltzmann Equation source code (DL_MESO_LBE) and the DL_MESO graphical user interface, and Richard Anderson at STFC Daresbury Laboratory for his contributions to the Dissipative Particle Dynamics source code (DL_MESO_DPD).

Chapter 1

DL_MESO General Information

1.1 Description

DL_MESO is a general purpose mesoscopic simulation package developed at Daresbury Laboratory by Dr Michael Seaton under the auspices of the Engineering and Physical Sciences Research Council (EPSRC) for the EPSRC's Collaborative Computational Project for the Computer Simulation of Condensed Phases (CCP5). The package is the property of the Science and Technology Facilities Council (STFC).

DL_MESO is issued free under licence to academic institutions pursuing scientific research of a non-commercial nature. All recipients of the code must first agree to the terms and conditions of the licence and register with us to be kept aware of new developments and discovered bugs. Commercial organisations interested in acquiring the package should approach the Computational Science and Engineering Department, STFC Daresbury Laboratory in the first instance. Daresbury Laboratory is the sole centre for distribution of the package. Under no account is it to be redistributed to third parties without consent of the owners.

DL_MESO contains two mesoscale simulation methods:

- Lattice Boltzmann Equation (included with version 1.0 and later)
- Dissipative Particle Dynamics (included with version 2.0 and later)

1.2 Functionality

The following is a list of the features that DL_MESO currently supports. Users are reminded that we are interested in hearing what other features could be usefully incorporated. We obviously have ideas of our own and CCP5 strongly influences developments, but other input would be welcome nevertheless.

1.2.1 Lattice Boltzmann Equation

DL_MESO_LBE can simulate lattice-gas systems using the Lattice Boltzmann Equation (LBE). The following properties and features are currently available:

- Multiple fluid components, solutes and coupled heat transfers[59]
- Collisions: Bhatnagar-Gross-Krook (BGK) single-relaxation-time[4] or Multiple-Relaxation-Time (MRT)[26, 7]
- Boundary conditions: Periodic, bounce-back (including stationary objects), constant pressure/velocity at planar surfaces[60]

- Mesoscale interactions: Shan-Chen pseudopotential method[43, 44]

1.2.2 Dissipative Particle Dynamics

DL_MESO_DPD can model DPD particles ('beads') with soft potential fields, by default the standard Groot-Warren potential[13], although Lennard Jones[23] and Weeks-Chandler-Andersen[57] potentials are also available, along with thermostating dissipative and random forces. The following properties and features are currently available:

- Choice of integrators/thermostats: standard Velocity Verlet, DPD Velocity Verlet[10], Lowe-Andersen[27], Peters[38] and Stoyanov-Groot[49]
- Constant volume (NVT) or constant pressure (NPT) simulations with Berendsen[2] or Langevin[22] barostats
- User selection of interaction lengths, conservative and dissipative force parameters for each species and between unlike species
- Bond stretching, angles and dihedrals between beads in user-defined 'molecules'
- Density-dependent conservative forces (many-body DPD)[37, 53]
- Electrostatic potentials between charged beads using a modified Ewald summation[11]
- Periodic or hard reflecting boundaries with optional short-range repulsions[56]

1.3 Requirements

1.3.1 Software requirements

- Standard C++ Compiler for LBE source code, DL_MESO_LBE
- Standard Fortran90 Compiler for DPD source code, DL_MESO_DPD
- Message Passing Interface (if parallel execution required)
- JAVA 2 Version 1.4 or higher (if GUI is to be used)
- GNU Make (included in standard Unix/Linux distributions; can be installed for Windows)

1.3.2 System requirements

DL_MESO is designed to work in both serial and parallel running; it can be run on standalone machines, clusters and supercomputers. The code has been tested on Solaris, Windows XP, IBM p690+ HPCx, PowerPC 450 Blue Gene/P and Cray XT4/XT6 HECToR machines.

1.4 The DL_MESO Directory Structure

The supplied version of DL_MESO is a gzipped tar file, which unpacks as directory **dl_meso_2.x**, where x is a generation number. Beneath the top level of this directory are a number of subdirectories:

- **LBE** - containing the LBE source code
- **DPD** - containing the DPD source code

- **JAVA** - containing the GUI source code
- **MAN** - containing the DL_MESO user manual
- **DEMO** - containing test cases for DL_MESO
- **WORK** - an example 'working directory'

1.5 Disclaimer

Neither STFC, CCP5 nor any of the authors of the DL_MESO package guarantee that the package is free from error. Neither do they accept responsibility for any loss or damage that results from its use.

1.6 Copyright

© STFC Daresbury Laboratory 2011

1.7 Authors

Dr Michael Seaton and Prof. William Smith
Computational Science & Engineering Department
STFC Daresbury Laboratory
Warrington
WA4 4AD
United Kingdom

1.8 Suggestions and Bug Reports

We encourage users to send suggestions for improvements and new features for DL_MESO, including bug reports and subroutines, as well as any additional test cases that demonstrate its features. All of these should be sent to `michael.seaton@stfc.ac.uk`

Chapter 2

The DL_MESO GUI

2.1 Getting Started with the DL_MESO GUI

The DL_MESO GUI offers a convenient way of using the DL_MESO package, although it is not an essential tool for those who prefer command line operation: Appendix A provides details on compiling the DL_MESO program codes manually. Working with the GUI requires the availability of Java tools, particularly the `javac` compiler and the `java` runner for Java 2 version 1.4 or later. These may be obtained from the `java.sun.com` website.

To build the GUI, proceed as follows:

- Enter the `DL_MESO/JAVA` directory.
- Type `javac *.java` to compile the source code.
- Type `jar -cfm GUI.jar manifest.mf *.class` to create the `GUI.jar` executable `JAR` file.
- Move to your working directory.
- Launch the GUI.

A Unix/Linux script called `makegui` that performs the build of the GUI can be found in the `JAVA` subdirectory.

Your working directory is the directory from which you wish to work when running DL_MESO. Working there will keep any files you generate separate from the DL_MESO source files. **Note** in the current version of DL_MESO the working directory should be at the same directory level as the `JAVA` direction, i.e. within the `DL_MESO` top directory, and contain the executables of any external utilities required to set up input files and gather or process output files from simulations. An example of such a working directory (called `WORK`) is present under the `DL_MESO` top directory; this includes a makefile to compile external utilties which can be invoked by the command `make -f Makefile-utils`.

In your working directory you can start the GUI with the command

- `java -jar ../JAVA/GUI.jar`

You may consider saving this command in a script for simple execution. An example script for Unix/Linux called `rungui` is present in the `WORK` subdirectory.

Figure 2.1 shows the DL_MESO GUI when it is started. Clicking the **LBE** and **DPD** buttons will produce the Lattice Boltzmann and Dissipative Particle Dynamics panels respectively, which will guide you through setting up input files, modifying and compiling the program code, running the simulation and gathering the results files

Figure 2.1: DL_MESO GUI on startup

for plotting and visualization. The **SPH** button is for Smoothed Particle Hydrodynamic simulations, which will be included in future versions of DL_MESO: clicking on this button will currently produce a warning message. This user manual can be read in Adobe Acrobat Reader (if installed) by clicking the **Manual** button, while **Help** will advise you to visit the DL_MESO website at `www.ccp5.ac.uk/DL_MESO`.

2.2 Lattice Boltzmann and the DL_MESO GUI

To access the LBE facilities in the DL_MESO GUI, proceed as follows:

- Click the **LBE** button to get the LBE panel.
- Click the **Define LBE System** button and supply the required information. The file `lbin.sys` will be created by the step.
- Click the **Set LBE Space** button to define the simulation space. The file `lbin.spa` will be created by this step.

2.2.1 Defining the System

Figure 2.2 shows the Define LBE System panel with the rows for data entry labelled in red numbering. The required data are as follows:

1. The required **LBE model** can be selected from the pull-down list: the D2Q9, D3Q15, D3Q19 and D3Q27 square lattice schemes are available.
2. The **number of grid points** sets the size of the system. For 2D systems, the number of grid points in the z direction must equal 1; selecting a two-dimensional lattice model greys out this box.
3. The **total steps** for the simulation and the **save span** (number of timesteps between system outputs) are given in this row.

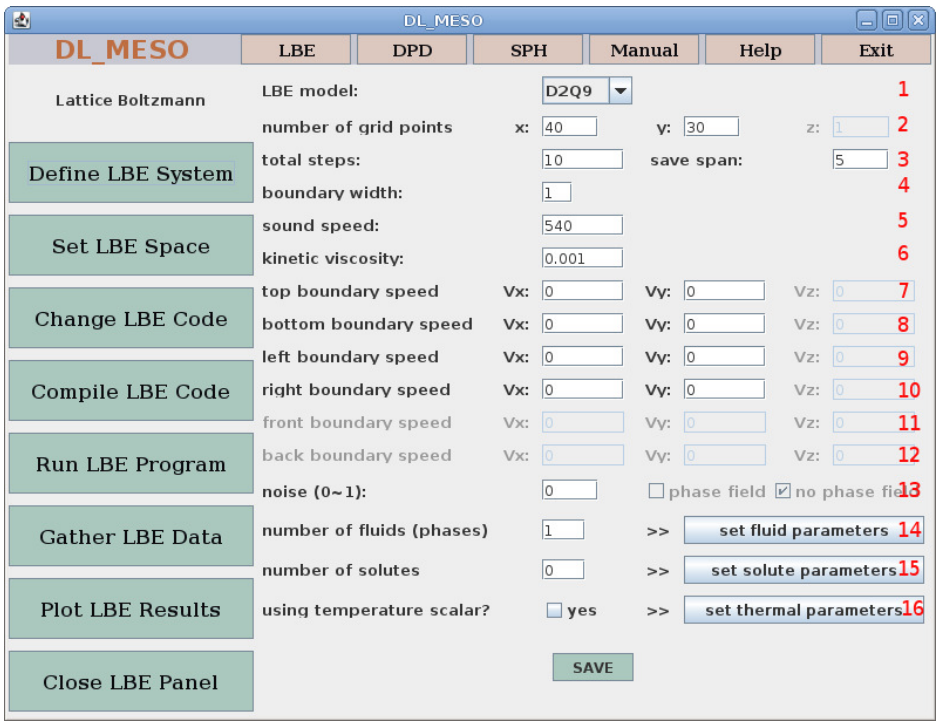


Figure 2.2: Define LBE System

4. The **boundary width** for running the parallel version of DL_MESO_LBE is set using this box. (The serial version by default automatically resets the boundary width to zero.)

5. The **sound speed** (c) is the real-life quantity for the first (main) fluid.

6. The **kinetic viscosity** (ν) is the real-life quantity for the first (main) fluid and, together with the sound speed, allows conversions between lattice and real units: the time step and lattice spacing are given by $\Delta t = \frac{\nu}{c^2\left(\tau_f - \frac{1}{2}\right)}$ and $\Delta x = \frac{\sqrt{3}\nu}{c\left(\tau_f - \frac{1}{2}\right)}$ respectively.

7. The specified **top boundary speed** (in lattice units), i.e. at the maximum y value for the grid. This and similar properties only need to be specified if the boundary includes a fixed velocity; any specified value will be ignored for periodic, bounce-back and fixed density conditions. Note that the z-component will be greyed out for two-dimensional systems.

8. The specified **bottom boundary speed** (in lattice units), i.e. at the minimum y value for the grid.

9. The specified **left boundary speed** (in lattice units), i.e. at the minimum x value for the grid.

10. The specified **right boundary speed** (in lattice units), i.e. at the maximum x value for the grid.

11. The specified **front boundary speed** (in lattice units), i.e. at the maximum z value for the grid. For two-dimensional systems this can be omitted and is thus greyed out.

12. The specified **back boundary speed** (in lattice units), i.e. at the minimum z value for the grid. For two-dimensional systems this can be omitted and is thus greyed out.

13. The **noise** magnitude only has an effect for initializing multiple phase simulations. DL_MESO_LBE may include either a **phase field** parameter or **no phase field** parameter for systems with multiple phases; the Shan/Chen[43] algorithm does not require it and this option is currently disabled.

14. The **number of fluids (phases)** can be increased if a multiple fluid system is to be studied: up to 6 fluids may be modelled in DL_MESO_LBE. The parameters and boundary conditions for the fluid(s) must then be set by clicking the **set fluid parameters** button – see below for more details.

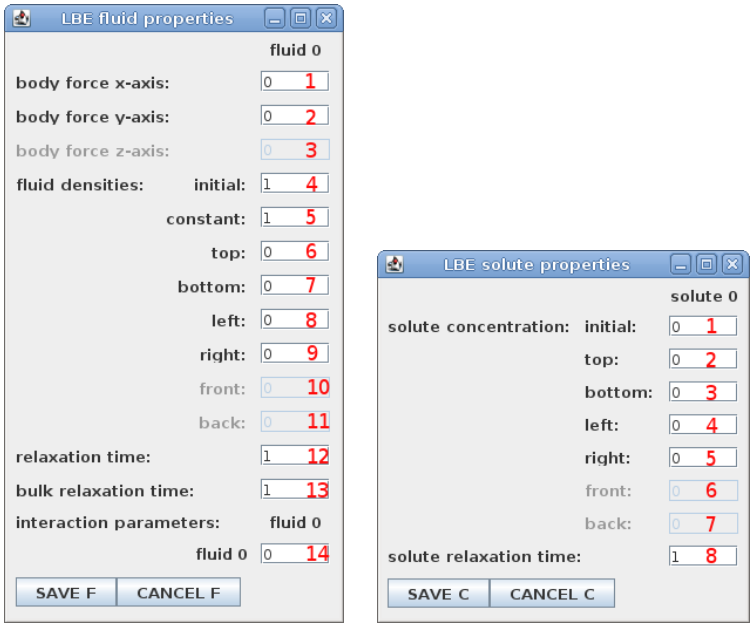
15. The **number of solutes** needs changing if solute parameters are required: if the number of solutes is greater than zero (and up to 6), the number of fluids in the above row must be set to 0 or 1. If used, the parameters and boundary conditions for the solutes must be set by clicking the **set solute parameters** button – see below for more details.

16. The **using temperature scalar** box may be clicked **yes** if thermal systems are to be studied. If checked, the thermal parameters must be set by clicking the **set thermal parameters** button – see below for more details.

Once all the data in this window and any pop-up windows for fluid, solute and thermal parameters are filled in, the **SAVE** button should be clicked to write the `lbin.sys` file.

2.2.1.1 Fluid, solute and thermal parameters

Examples of the pop-up windows for fluid, solute and thermal parameters can be seen in Figure 2.3 with the rows labelled in red numbering: multiple columns of dialogue boxes are made available for systems with multiple fluids and/or solutes.



(a) Fluid parameters (b) Solute parameters

(c) Thermal parameters

Figure 2.3: Fluid, solute and thermal parameter pop-up windows

For fluids, the required data are as follows:

1. **Body force x-axis**: the x-component of body force on each fluid.

2. **Body force y-axis**: the y-component of body force on each fluid.

3. **Body force z-axis**: the z-component of body force on each fluid. (Greyed out for two-dimensional systems.)
4. The **initial** fluid densities are applied throughout the system and used to initialize LBE calculations.
5. The **constant** fluid density (ρ_0) for incompressible systems.
6. The fluid densities at the **top** boundary.
7. The fluid densities at the **bottom** boundary.
8. The fluid densities at the **left** boundary.
9. The fluid densities at the **right** boundary.
10. The fluid densities at the **front** boundary. (Greyed out for two-dimensional systems.)
11. The fluid densities at the **back** boundary. (Greyed out for two-dimensional systems.)
12. The **relaxation time** (τ_f) for each fluid: these values should be greater than 0.5 to give non-zero kinetic viscosities.
13. The **bulk relaxation time** ($\tau_{f,bulk}$) for each fluid: these values are only used in multiple-relaxation-time schemes to define the bulk viscosity and again should be greater than 0.5.
14. If more than one phase is being modelled, non-zero **interaction parameters** between the fluids are required (g_{ab} for the Shan/Chen algorithm).

Solutes require the following data in the following row numbers:

1. The **initial** concentrations of the solutes throughout the system, as used for initialization.
2. The solute concentrations at the **top** boundary.
3. The solute concentrations at the **bottom** boundary.
4. The solute concentrations at the **left** boundary.
5. The solute concentrations at the **right** boundary.
6. The solute concentrations at the **front** boundary. (Greyed out for two-dimensional systems.)
7. The solute concentrations at the **back** boundary. (Greyed out for two-dimensional systems.)
8. The **relaxation time** (τ_s) for each solute, representing diffusivities.

If selected for inclusion, the required thermal properties are:

1. The **heat relaxation time** (τ_t) for the system, which represents the thermal diffusivity.
2. The **initial T(K)** and **initial dT/dt**: initial temperature and rate of change in temperature (related to heat transfers to/from the system) for the entire system.
3. The temperatures and rates of temperature change at the **top** and **bottom** boundaries.
4. The temperatures and rates of temperature change at the **left** and **right** boundaries.
5. The temperatures and rates of temperature change at the **front** and **back** boundaries. (These are greyed out for two-dimensional systems.)

After filling in all the required values, clicking the relevant save button (**SAVE F**, **SAVE C** or **SAVE T**) will store the data in preparation for writing to the `lbin.sys` input file. The cancel buttons (**CANCEL F**, **CANCEL C** and **CANCEL T**) will close the pop-ups without saving any values.

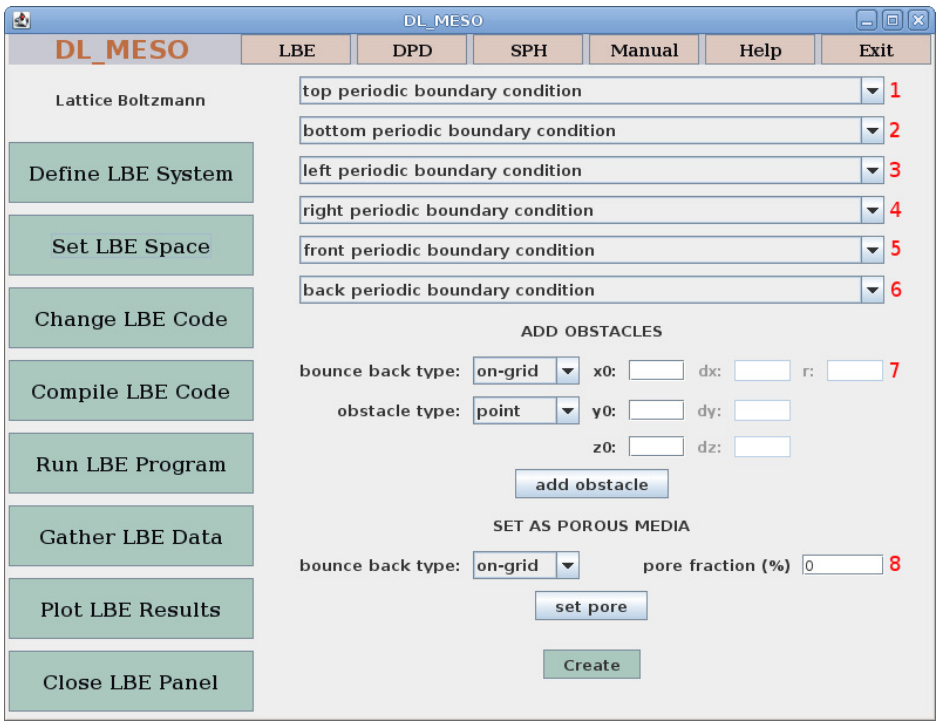


Figure 2.4: Set LBE Space

2.2.2 Defining the Space Properties

If this option is selected before saving the LBE system data, a warning message advising that the system should be re-defined will appear.

Figure 2.4 shows the Set LBE Space panel with the rows for data entry labelled in red numbering. The following data are required:

1. The **top boundary condition** can be selected using the pull-down list from:
   - periodic
   - on-grid bounce back
   - mid-grid bounce back
   - fixed V (velocity), C (concentration) and T (temperature)
   - fixed V and C, Neumann[1] T
   - fixed V and T, Neumann C
   - fixed V, Neumann C and T
   - fixed P (pressure or density), C and T
   - fixed P and T, Neumann C
   - fixed P and C, Neumann T
   - fixed P, Neumann C and T
2. The **bottom boundary condition** can be selected using the pull-down list.
3. The **left boundary condition** can be selected using the pull-down list.
4. The **right boundary condition** can be selected using the pull-down list.

[1]For a property ϕ, DL_MESO currently only calculates $\nabla\phi = 0$ by using on-grid bounce back on the related distribution function.

5. The **front boundary condition** can be selected using the pull-down list. This pull-down list will be greyed out for two-dimensional systems.

6. The **back boundary condition** can be selected using the pull-down list. This pull-down list will be greyed out for two-dimensional systems.

7. Solid obstacles can be added to the calculation space by selecting the bounce back (on-grid or mid-grid) and obstacle types in the pull-down lists, entering its location on the grid and, if necessary, entering its size, and clicking **add obstacle**.

   - A single **point** will be located at (**x0**, **y0**, **z0**).
   - A **sphere** is centred at (**x0**, **y0**, **z0**) and has radius **r**.
   - A two-dimensional **circle** is centred at (**x0**, **y0**) and has radius **r**.
   - A **block** has a vertex at (**x0**, **y0**, **z0**) and has size (**dx**, **dy**, **dz**): both **z0** and **dz** can be omitted for two-dimensional blocks.

   Note that lattice points well within an obstacle are set as blank sites, i.e. they will be ignored in LBE calculations.

8. The entire system can be set up as a porous solid by selecting the bounce back type, specifying a **pore fraction** and clicking **set pore** to randomly select an appropriate number of solid lattice sites.

Clicking the **Create** button will write all the lattice space data to a `lbin.spa` file; any lattice point defined more than once will hold its *latest* definition.

2.3 Dissipative Particle Dynamics and the DL_MESO GUI

To access the DPD facilities in the DL_MESO GUI, proceed as follows:

- Click the **DPD** button to get the DPD panel.
- Click the **Define DPD System** button and supply the required information. The `CONTROL` file will be created by this step.
- Note that currently no simulation space settings or molecular structure data can be entered using the GUI.
- Click **EXIT** to finish the settings.

2.3.1 Defining the System

Figure 2.5 shows the Define DPD System panel with the rows for data entry labelled in red numbering.

The required data are as follows:

1. The **job header**: a line of text up to 80 characters long describing the simulation.

2. The system **volume**, which can either be cubic or orthogonal (cuboidal) as selected from the pull-down list. If specifying a cubic volume, the total volume should be specified, while orthogonal volumes require the sizes for all three dimensions.

3. The target **temperature** (k_BT) and **pressure** (P_0) for the system. (The latter is greyed out if no barostat is to be used.)

4. The **dissipative factor** (γ for DPD and Peters thermostats, Γ for Lowe-Andersen and Stoyanov-Groot thermostats) for the entire system – values for individual species and between unlike species can be specified elsewhere – and the size of the **boundary halo** for copying particle data from neighbouring subdomains or across periodic boundaries.

5. The **interaction cutoff** (r_c) for pairwise particle interactions and the **many-body cutoff** (r_d) for determining localized particle densities as used by many-body DPD. (The latter is greyed out if not in use.)

6. If required, the **electrostatic cutoff** (r_e) for short-range electrostatic interactions and the **surface cutoff** (z_c) for interactions between particles and solid walls.

7. The size of each **time step** (Δt) for integrating the equations of motion and the **total steps** required for the DPD simulation.

8. The number of time steps required to equilibrate the system (**equilibration steps**) and the number of time steps to store system variables for rolling averages (**stack interval**).

9. The numbers of time steps between rescaling of particle velocities to the desired system temperature during equilibration (**temp scaling interval**) and printing summaries in the `OUTPUT` file (**print interval**). The former can be set to zero if no temperature rescaling is required.

10. The numbers of time steps between saves of trajectory data to `HISTORY` files for later visualization (**save interval**) and between outputs of statistical data (system energy, potential energies, pressure, temperature etc.) to a plottable `CORREL` file. These values can respectively be set to zero if no trajectory or plot files are required.

11. The **job time** is the maximum (real) time that can be spent carrying out the DPD simulation: the **close time** gives the time needed to write restart files and shut down the calculation in a controlled manner.

12. The **restart key** for the simulation: this can either be set to **none** for a new simulation, a **full restart** to continue a previous run using `export*` files, a **new run** which takes a starting state (particle positions and velocities) for a new simulation from `export*` files, and **rescaled** does the same as a new run but additionally rescales the particle velocities to give the specified system temperature.

13. The **number of species** involved in the simulation: a maximum of 6 species can be specified in DL_MESO_DPD and defined using the GUI. The properties for the species must then be set by clicking on **set species** – see below for more details. The parameters for pair potentials, thermostats, barostats, electrostatics and surfaces *can only be set once those for the species are defined.*

14. The **potential key** gives the definition of potential energy functions used in calculations: the standard DPD model by Groot and Warren[13] is the default, but many-body (density dependent) DPD[37, 53], Lennard-Jones[23] and Weeks-Chandler-Anderson 'hard sphere' models can also be selected. Note that the Lennard-Jones and Weeks-Chandler-Anderson[57] models are non-stochastic and *not DPD* models, although the DPD thermostat can be used with them to maintain system temperature. Parameters for unlike species interactions (if more than one species is specified) and for many-body DPD interactions can be set by clicking on **set potentials** – see below for more details.

15. The system **thermostat**: the dissipative and random forces as defined for DPD with the standard (molecular dynamics) form of the Velocity Verlet integrator (DPD/MD-VV) is the default, but recalculation of dissipative forces at the end of each step (DPD/DPD-VV)[10], the Lowe-Andersen[27], Peters[38] and Stoyanov-Groot[49] thermostats can also be selected. Values of γ for the DPD and Peters thermostats and Γ for the Lowe-Andersen and Stoyanov-Groot thermostats can be specified elsewhere for the entire system, individual species or between unlike species, but an additional parameter for the Stoyanov-Groot thermostat should be set by clicking on **set thermostat** – see below for more details.

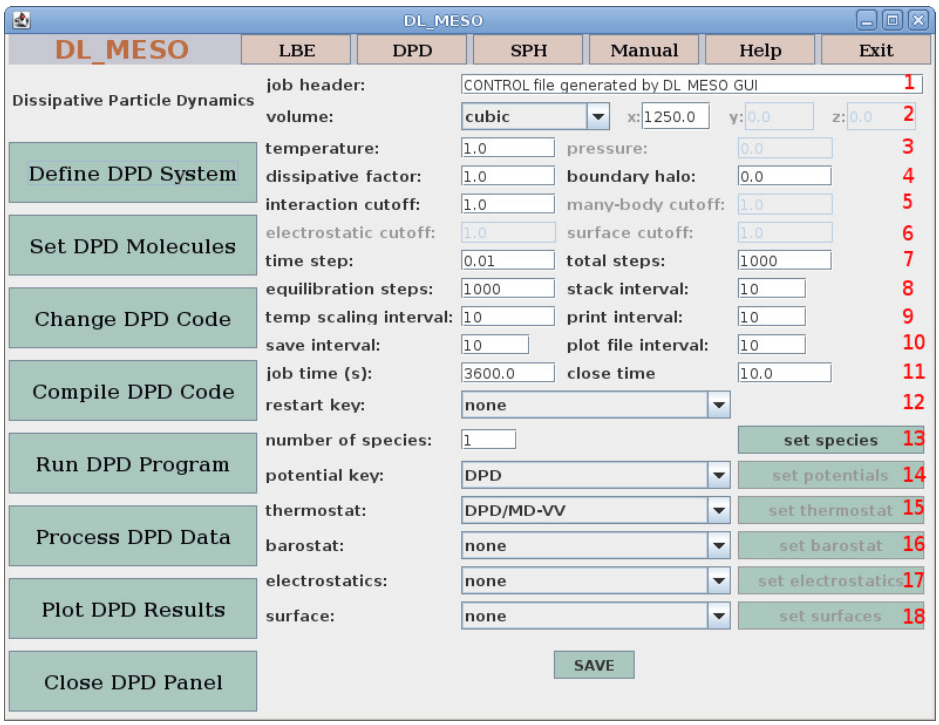


Figure 2.5: Define DPD System

16. The system **barostat**: no barostat is used by default, but Langevin[22] and Berendsen[2] barostats are available in combination with all five thermostats. If either barostat is selected, its parameters can be set by clicking on **set barostat** and the target system pressure can be specified.

17. The **electrostatics** scheme for the simulation: an Ewald sum method with Slater-type (exponential) charge smearing[11] is available in DL_MESO_DPD. If selected, the short-range electrostatic cutoff can be edited and the parameters for the Ewald sum can be specified by clicking on **set electrostatics**.

18. The **surfaces** to be applied to the system: by default periodic boundary conditions are used, but hard walls can be applied at the boundaries with soft repulsions and specular reflection[56]. Selecting this method allows the surface cutoff to be edited and the wall parameters can be specified by clicking on **set surfaces**.

Note that the DPD code uses reduced units in which the unit of length is the particle size, the unit of mass is the particle mass and the unit of energy is the primary energy parameter of the potential energy function. From these the time unit may be derived. The temperature is defined to be $\frac{2}{3}$ of the system kinetic energy.

The `CONTROL` file for input into DL_MESO_DPD is created by clicking the **SAVE** button.

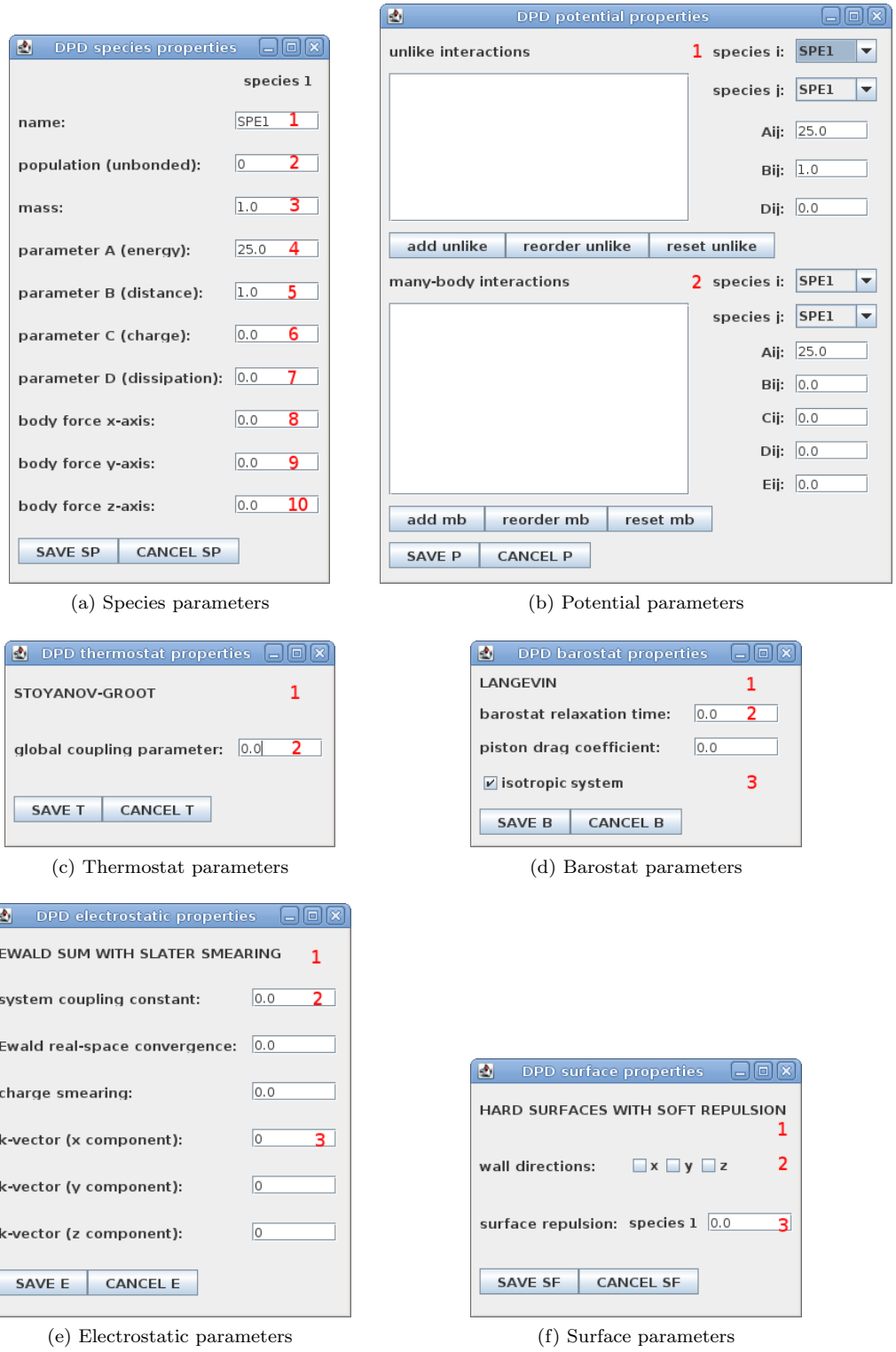
2.3.1.1 Species, potential, thermostat, barostat, electrostatic and surface parameters

Examples of the pop-up windows for species, potential, thermostat, barostat, electrostatic and surface parameters can be seen in Figure 2.6 with the rows labelled in red numbering: multiple columns of dialogue boxes are made available for systems with multiple species.

For species, the required data are as follows (referring to Figure 2.6(a)):

1. Species **name**: this can be up to 8 alphanumeric characters (overly long names will be truncated).

2. Species **population**: the number of *unbonded* particles of the species in the system. (If molecules are to be included, species that do not exist in unbonded form should still be defined here, but the unbonded population should be set to zero.)



(a) Species parameters

(b) Potential parameters

(c) Thermostat parameters

(d) Barostat parameters

(e) Electrostatic parameters

(f) Surface parameters

Figure 2.6: Species, potential, thermostat, barostat, electrostatic and surface pop-up windows

3. Particle **mass** for the species.

4. **Parameter A (energy)**: this parameter is equal either to the conservative interaction parameter ($A_i \equiv A_{ii}$) for DPD beads or the potential well depth ($\epsilon_i \equiv \epsilon_{ii}$) for hard sphere systems (i.e. Lennard-Jones, Weeks-Chandler-Anderson).

5. **Parameter B (distance)**: this parameter is equal either to the particle 'radius' (B_i) for DPD or zero-potential distance (σ_i) for hard sphere models.

6. **Parameter C (charge)**: this parameter is equal to the net charge of the particle relative to the charge on an electron (e). This parameter does not have to be specified for systems without electrostatic interactions.
7. **Parameter D (dissipation)**: this parameter is equal either to the dissipative factor (γ_i) as used by the DPD and Peters thermostats, or to the collision frequency (Γ_i) for the Lowe-Andersen and Stoyanov-Groot thermostats. If this value is not specified or set to zero, the value for the entire system will apply.
8. **Body force x-axis**: x-component of constant external body force on each particle.
9. **Body force y-axis**: y-component of constant external body force on each particle.
10. **Body force z-axis**: z-component of constant external body force on each particle.

Potentials require the following data (see Figure 2.6(b)):

1. The energy (**Aij**), distance (**Bij**) and dissipation (**Dij**) parameters between species i and j. If any of these values are set to zero, DL_MESO_DPD will use default values, i.e. $A_{ij} = \sqrt{A_i A_j}$, $B_{ij} = \frac{1}{2}(B_i + B_j)$, $\gamma_{ij} = \gamma$. Clicking **add unlike** will store the entered parameters to memory and add a record to the text box (which cannot be edited by the user), **reorder unlike** will reorder the specified interactions and **reset unlike** will clear all of them.
2. The many-body DPD parameters (**Aij**, **Bij**, **Cij**, **Dij** and **Eij**) between species i and j. Parameters for all possible species pairs *must* be specified by the user. Clicking **add mb** stores the entered parameters to memory and add a record to the text box, **reorder mb** reorders the specified interactions and **reset mb** will clear all many-body DPD parameters.

The thermostat pop-up window is formatted as in Figure 2.6(c):

1. The type of thermostat to be used in the simulation.
2. Thermostat parameters: for the Stoyanov-Groot thermostat (currently the only type that requires an additional parameter), a **global coupling parameter** for the Nosé-Hoover part (α) is required.

Figure 2.6(d) gives the layout for the barostat pop-up window:

1. The type of barostat to be used in the simulation.
2. Barostat parameters: for the Langevin barostat, a **barostat relaxation time** (τ_p) and **piston drag coefficient** (γ_p) are required, while the Berendsen barostat requires the **compressibility/relaxation** ratio ($\frac{\beta}{\tau_p}$).
3. This check box determines whether or not an **isotropic system**, i.e. one where pressure acts uniformly in all dimensions, should be modelled. If unchecked, the barostat will act differently in each dimension and the shape of the system will change over time.

The parameters for electrostatics can be given in the pop-up window shown in Figure 2.6(e):

1. The type of electrostatics to be used in the simulation.
2. Electrostatic parameters: for the Ewald sum with Slater-type smearing, the **system coupling constant** (Γ), **Ewald real-space convergence** (α) and **charge smearing** (β) coefficients need to be specified.
3. The Ewald sum method also requires a reciprocal space (**k-vector**) range.

If non-periodic boundaries are to be used, the parameters for surfaces can be entered in the appropriate pop-up window (Figure 2.6(f)):

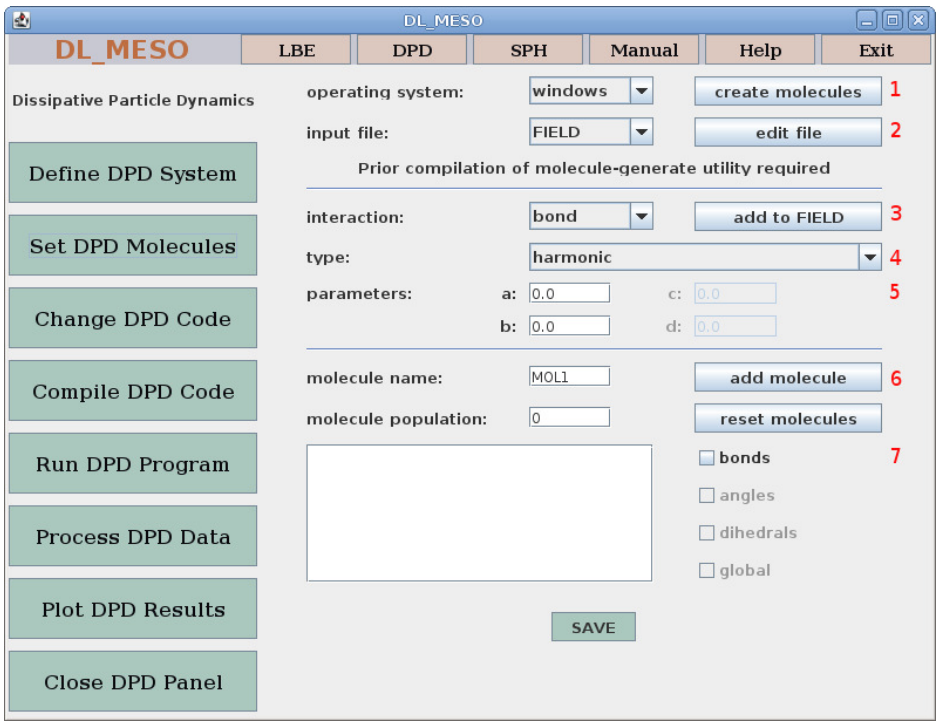


Figure 2.7: Set DPD Molecules

1. The type of surface interactions or boundary conditions to be applied.
2. **Wall directions**: if the checkbox for a particular dimension is ticked, the boundary condition will be applied to the surfaces orthogonal to the specified axis.
3. Surface parameters: for the hard surfaces with soft repulsions, the **surface repulsion** for each species ($A_{wall,i}$) is required.

After filling in all the required values, clicking the relevant save button (**SAVE SP**, **SAVE P**, **SAVE T**, **SAVE B**, **SAVE E** or **SAVE SF**) will store the data in preparation for writing to the `CONTROL` file. The cancel buttons (**CANCEL SP**, **CANCEL P**, **CANCEL T**, **CANCEL B**, **CANCEL E** and **CANCEL SF**) will close the pop-ups without saving any values.

2.3.2 Defining DPD Molecules

Figure 2.7 shows the Set DPD Molecules panel with the sections for data entry labelled in red numbering. The following data are required:

1. The **operating system** is required before launching a command-line terminal and running the molecule generation utility `molecule-generate.cpp` to **create molecules** for the DPD simulation. The utility should be compiled beforehand to give the executable `molecule.exe` (refer to Appendix B for more details).
2. Each **input file** for molecules, `FIELD` and `MOLECULE`, can be examined and/or edited using DL_MESO's built-in editor, which is launched by the button **edit file**.
3. In addition to identifying the particles with a molecule that are bonded together and, where necessary, make up any angles or dihedrals, the `FIELD` file also requires the parameters for each bond **interaction**. The pull-down list allows the user to select *bond*, *angle* or *dihedral* interactions.
4. The interaction **type** is selected using this pull-down list, whose contents are dependent on whether a bond, angle or dihedral is required.

5. The **parameters** are entered in these boxes, corresponding to Tables 10.4, 10.5 and 10.6 for bonds, angles and dihedrals respectively. The button **add to FIELD** will add the interaction and its parameters to the existing `FIELD` file.

6. The **molecule name** (of up to 8 characters in length) and **molecule population** for each molecule described in the input files is required in the `CONTROL` file: clicking **add molecule** stores the entered information in memory in the order it will appear in the `CONTROL` file, while **reset molecules** removes them all from memory. By default a maximum of 8 molecule types can be included in DL_MESO_DPD.

7. The types of bonded interactions can be selected and specified in the `CONTROL` file using these checkboxes: **bond** needs to be ticked if any molecules are to be included, while **angle** and **dihedral** only need to be selected if those interactions are required for any of the molecules being modelled. The **global** option makes DL_MESO_DPD store all bond data for the system in all processing units instead of (by default) storing bond data in each processing unit for only the bonds within its subdomain.

After the molecule name, populations and types of interactions required are entered, clicking **SAVE** will write this information to the existing `CONTROL` file.

2.4 Compiling and running DL_MESO

- Compiling the LBE/DPD code may be accomplished through the compiler panel which is activated from either of the **Compile LBE Code** or **Compile DPD Code** buttons.

- The **Compile LBE Code** panel allows you to select the operating system, a C++ compiler, compiler flags and the version (serial or parallel) of the code you wish to build. If you require a C++ compiler that is not included in the pull-down list, select other and type the command for the required compiler in the neighbouring box. Clicking the **COMPILE** button will start the compilation and a message box will signal its completion.

- The **Compile DPD Code** panel allows you to select the operating system, a Fortran90 compiler, compiler flags and the version (serial or parallel) of the code you wish to build. If you require a Fortran90 compiler that is not included in the pull-down list, select other and type the command for the required compiler in the neighbouring box. The **Create Makefile** button needs to be clicked first to create a makefile in the working directory, which automates compilation and may be edited by the user. Clicking the **COMPILE** button will invoke the makefile to compile the code and a message box will signal its completion.

- If the compilation fails, you may need to edit the code. An editing panel is available for this purpose using either the **Change LBE Code** or **Change DPD code** buttons. Its function is similar to the compilation panel in operation with a choice of text editors, including one packaged with the DL_MESO GUI.

  - The files in the LBE code that can be edited include the parallel and serial main files, the lattice model file, the boundary condition file, the core routines for LBE calculations, the file for user-defined routines, the main head file and the head file for user-defined routines. Others can be edited by selecting other and typing the name of the file in the neighbouring box.
  - The files in the DPD code that may be edited include the main program, constants, global variables and the modules configuration, start (for system initialization), field (for force calculations), bond interactions, many-body DPD, surfaces and statistics. Other code files can be edited by selecting other and typing the name in the neighbouring box.

- Running the LBE/DPD code is made possible through the **Run LBE Program** or **Run DPD Program** button, which activates a panel that allows you to select the required submission command and then submit the job. You may need to create a suitable run script in your working directory beforehand if running the job in parallel.

- Collecting data from multiple processors and processing it for visualization is possible using the **Gather LBE Data** and **Process DPD Data** buttons. Note that the utilities need to be compiled in the working directory prior to use: details on this and their functions can be found in Appendix B.
- The results of LBE and DPD calculations may be plotted using the **Plot LBE Results** and **Plot DPD Results** buttons, which allows the user to select plotting and visualization applications, including those not available in the pull-down lists. Note that these need to be already installed on the workstation in use before being invoked by the GUI: if they require running from a command-line, tick the **run in terminal** box before launching the application.

2.5 Notes

- There are some inactive buttons reserved for later use.
- The GUI does not produce initial state files (`lbin.init` for LBE, `CONFIG` for DPD) prior to simulations.
- Click **EXIT** to close down the GUI.

Part I

Lattice Boltzmann Equation (LBE)

Chapter 3

The Lattice Boltzmann Equation: Basic Theory

3.1 Introduction

The Lattice Boltzmann method utilizes fully discretized space, time and velocity to describe the evolution of fluid. Space is represented by a regularly distributed grid. Time flow is obtained by integrating over discrete time steps. Discrete velocity vectors are defined to ensure that a particle moves from one grid point to another without falling between grid points.

The Lattice Boltzmann method inherits the discrete concept from its ancestor, Lattice Gas Cellular Automata (LGCA). The main difference between the Lattice Boltzmann Equation (LBE) method and LGCA is that LBE describes the physical state of a grid by a set of single particle distribution functions instead of a particle number. This allows LBE to simulate condensed matter fluids distinct from diluted fluids in which the mean free path of the component particles is much larger than the lattice distance.

The Lattice Boltzmann algorithm can be summarized by the following:

- Fluid properties are mapped onto a discrete lattice.
- The physical state at each lattice point is described by a set of particle distribution functions.
- Macroscopic fluid variables are defined by moments of the distribution functions.
- The system evolution towards equilibrium is through the relaxation of the distribution function to its equilibrium form.

3.2 Basic Definitions

Triangular and rectangular lattices are two of the most popular grid forms. The former have O6 rotation isotropy and have been widely applied in two-dimensional systems, e.g. the D2Q7 and D2Q13 models. The latter have only O4 rotation isotropy but more easily handle the simulation of three-dimensional systems with complex boundary conditions. The equilibrium for D2Q9 and D3Q27 lattice models can be derived *a priori* from the Maxwell equilibrium distribution. D3Q15 and D3Q19 models appear to be more popular than D3Q27 because the latter is much more expensive in terms of computing cost.

It is required that the equilibrium state should be able to reproduce elementary macroscopic fluid variables:

$$\rho = \sum_{i=0}^{q} f_i \tag{3.1}$$

$$\rho u_\alpha = \sum_{i=0}^{q} f_i e_{i\alpha} \tag{3.2}$$

where ρ is the density, f_i the ith particle distribution function, $\hat{e}_i$ the ith speed vector and u_α the macroscopic velocity along the α-axis. In the Lattice Boltzmann method, $\hat{e}_i$ is not the thermal velocity of a particle and therefore

$$E \neq \frac{1}{2} \sum f_i \left(\hat{e}_i - \vec{u}\right)^2 \tag{3.3}$$

Equation (3.3) implies that the temperature cannot be derived from the lattice particle distribution function. However, temperature *can* be defined at each grid point.

3.3 Derivation of Equilibrium

There are two methods by which the Lattice Boltzmann equilibria can be constructed. The *bottom-up* method obtains the equilibrium from the Maxwell-Boltzmann equilibrium distribution. The *top-down* method constructs the equilibrium so that the required macroscopic properties can be reproduced. Only the bottom-up method is shown here; the top-down method can be found in [5].

The Maxwell-Boltzmann single particle equilibrium distribution function is

$$f^{eq} = \frac{\rho}{(2\pi\theta)^{\frac{D}{2}}} \exp\left[-\frac{\left(\vec{\xi} - \vec{u}\right)^2}{2\theta}\right] \tag{3.4}$$

where $\theta = k_B T/m$, k_B is the Boltzmann constant, T is temperature, m is molar mass, D is the space dimension, $\vec{\xi}$ is the thermal velocity and $\vec{u}$ the macroscopic velocity.

When $|\vec{\xi} - \vec{u}| \ll \sqrt{\theta}$, Equation (3.4) can be expanded into

$$f^{eq} = \frac{\rho}{(2\pi\theta)^{\frac{D}{2}}} \exp\left(-\frac{\xi^2}{2\theta}\right)\left[1 + \frac{\vec{\xi}\cdot\vec{u}}{\theta} + \frac{\left(\vec{\xi}\cdot\vec{u}\right)^2}{2\theta^2} - \frac{u^2}{2\theta}\right] \tag{3.5}$$

For a microscopic quantity $\psi(\xi)$, the associated macroscopic quantity Ψ is calculated by

$$\Psi = \int \psi(\xi) f^{eq} d\xi \tag{3.6}$$

Let $\vec{\xi} = \sqrt{2\theta}\vec{c}$, where $\vec{c}$ is a rescaled thermal velocity; the macroscopic velocity $\vec{u}$ can be similarly rescaled to $\sqrt{2\theta}\vec{u}$. Equations (3.5) and (3.6) can thus be combined to give

$$\Psi = \int e^{-c^2} \psi(c) \frac{\sqrt{2\theta}\rho}{(2\pi\theta)^{\frac{D}{2}}}\left[1 + 2\left(\vec{c}\cdot\vec{u}\right) + 2\left(\vec{c}\cdot\vec{u}\right)^2 - u^2\right] dc \tag{3.7}$$

Using Gaussian quadrature, Equation (3.7) changes into

$$\Psi = \sum_i \psi(c_i) \frac{\sqrt{2\theta}\rho}{(2\pi\theta)^{\frac{D}{2}}} w(c_i)\left[1 + 2\left(\vec{c}\cdot\vec{u}\right) + 2\left(\vec{c}\cdot\vec{u}\right)^2 - u^2\right] \tag{3.8}$$

Let

$$w_i = \frac{\sqrt{2\theta}\rho}{(2\pi\theta)^{\frac{D}{2}}} w(c_i) \tag{3.9}$$

and

$$f_i^{eq} = w_i \rho\left[1 + 2\left(\vec{c}\cdot\vec{u}\right) + 2\left(\vec{c}\cdot\vec{u}\right)^2 - u^2\right] \tag{3.10}$$

The value of $w(c_i)$ can be obtained from Gauss-Hermite integration. Equation (3.10) is the equilibrium particle distribution function in the discrete regime. w_i is called the weight factor for speed vector c_i. Equation (3.10) can also be written as

$$f_i^{eq} = w_i \rho \left[1 + \frac{3(\hat{e}_i \cdot \vec{u})}{c^2} + \frac{9(\hat{e}_i \cdot \vec{u})^2}{2c^4} - \frac{3u^2}{2c^2} \right] \tag{3.11}$$

where $c = \sqrt{3\theta} = \sqrt{\frac{3k_B T}{m}}$ is the modulus of the basic lattice vector. For water at 20℃, $c = 367.8$ m/s.

3.4 Structural Relaxation and Macroscopic Equations

The Lattice Boltzmann method often uses the BGK (Bhatnager, Gross and Krook) approximation[4] to describe the structural relaxation. The single particle distribution function evolves to the equilibrium state via

$$f_i(\vec{x} + \hat{e}_i \Delta t, t + \Delta t) - f_i(\vec{x}, t) = -\frac{\Delta t}{\tau_f} \left[f_i(\vec{x}, t) - f_i^{eq} \right] \tag{3.12}$$

where τ_f is called the relaxation time and is related to the kinetic viscosity of fluid.

To derive the macroscopic equations, the left hand side of Equation (3.12) can be expanded as

$$f_i(\vec{x} + \hat{e}_i \Delta t, t + \Delta t) - f_i(\vec{x}, t) = \sum_{m=1}^{\infty} \frac{\Delta t^m}{m!} \left(\partial_t + e_{i\alpha} \partial_\alpha \right)^m f_i(\vec{x}, t) \tag{3.13}$$

Expanding the instantaneous particle distribution function around its equilibrium and retaining only the first order gives

$$f_i(\vec{x}, t) = f_i^{eq}(\vec{x}, t) - \tau_f \left(\partial_t + e_{i\alpha} \partial_\alpha \right) f_i^{eq}(\vec{x}, t) + O\left(\partial^2\right) \tag{3.14}$$

Substituting Equations (3.13) and (3.14) into the left hand side of Equation (3.12) gives the second order differential equation for the equilibrium distribution

$$\frac{f_i^{eq} - f_i}{\tau_f} = \left(\partial_t + e_{i\alpha} \partial_\alpha \right) f_i^{eq} - w_f \left(\partial_t + e_{i\alpha} \partial_\alpha \right)^2 f_i^{eq} + O\left(\partial^3\right) \tag{3.15}$$

where $w_f = \tau_f - \frac{\Delta t}{2}$.

Summing Equation (3.15) over i and ignoring the second order deriviative we obtain

$$0 = \partial_t \rho + \partial_\alpha \rho u_\alpha - w_f \partial_\beta \left(\partial \rho u_\beta + \partial_\alpha \sum_i f_i^{eq} e_{i\alpha} e_{i\beta} \right) + O\left(\partial^3\right) \tag{3.16}$$

Summing Equation (3.16) times e_i over i we obtain

$$0 = \partial_t \rho u_\alpha + \partial_\beta \sum_i f_i^{eq} e_{i\alpha} e_{i\beta} - w_f \partial_\gamma \left(\partial_t \sum_i f_i^{eq} e_{i\alpha} e_{i\gamma} + \partial_\beta \sum_i f_i^{eq} e_{i\alpha} e_{i\beta} e_{i\gamma} \right) + O\left(\delta^3\right) \tag{3.17}$$

Equation (3.17) shows that the second term in Equation (3.16) is of the third order in the derivative. Therefore we have the continuity equation to the second order of the derivative

$$\partial_t \rho + \nabla \cdot \rho \vec{u} = 0 \tag{3.18}$$

Defining the third and fourth order moments

$$\sum_i f_i^{eq} e_{i\alpha} e_{i\beta} = P_{\alpha\beta} + \rho u_\alpha u_\beta \tag{3.19}$$

$$\sum_i f_i^{eq} e_{i\alpha} e_{i\beta} e_{i\gamma} = P_{\alpha\beta} u_\gamma + P_{\alpha\gamma} u_\beta + P_{\beta\gamma} u_\alpha + \rho u_\alpha u_\beta u_\gamma \tag{3.20}$$

With these definitions, Equation (3.17) leads to the weakly compressible Navier-Stokes equation

$$\partial_t(\rho u_\alpha) + \partial_\beta(\rho u_\alpha u_\beta) = -\partial_\beta P_{\alpha\beta} + \frac{w_f D}{3}\partial_\beta\left(\frac{\delta_{\alpha\beta} - 3\partial_\alpha P_{\alpha\beta}}{D}\partial_\gamma(\rho u_\gamma) + \partial_\alpha(\rho u_\beta) + \partial_\beta(\rho u_\alpha)\right) + O\left(\partial^3\right) \quad (3.21)$$

where the kinetic viscosity is given by $\nu = \frac{w_f}{3} = \frac{1}{3}\left(\tau_f - \frac{1}{2}\right)$.

Similarly, the convection diffusion equation governing the evolution of solute transfer and the convection/conduction equation governing the evolution of thermal transfer can be obtained.

3.5 Mesoscale Interaction

The rate of change of momentum is proportional to the force

$$\rho u_\alpha\left(t + \Delta t\right) = \rho u_\alpha\left(t\right) + F_\alpha \Delta t \quad (3.22)$$

The Lattice Boltzmann Equation takes into account the effect of relaxation and expresses Equation (3.22) in a new format[32]:

$$\rho u_\alpha\left(t + \Delta t\right) = \rho u_\alpha\left(t\right) + F_\alpha \tau_f \quad (3.23)$$

$\vec{F}$, with F_α as the component in the α direction, can be a long-ranged body force or any local interactions. In the Shan-Chen model for multiple phases and components[43], the force on component a is defined as

$$\vec{F}^a = -\psi^a\left(\vec{x}\right)\sum_b g_{ab}\sum_i w_i \psi^b\left(\vec{x} + \hat{e}_i\right)\hat{e}_i \quad (3.24)$$

where g_{ab} is the interaction coefficient between elements a and b. ψ^a is related to the density of element a and can take many different forms, e.g.

$$\psi^a\left(\vec{x}\right) = 1 - \exp\left(-\rho^a\right) \quad (3.25)$$

3.6 Summary of Lattice Boltzmann Equation

Lattice Boltzmann is an established numerical methodology for handling hydrodynamics in fluid. It is particularly suited to simulating systems with complex boundary conditions and is also suitable for systems with phase transitions since mesoscale interactions can be merged into the method easily.

Chapter 4

DL_MESO_LBE Basic Definition

4.1 Lattice models

DL_MESO_LBE utilizes a right-handed Cartesian coordinate system with the x-axis from left to right in the horizontal direction, the y-axis from low to high in the vertical direction and z-axis from back to front.

D2Q9, D3Q15, D3Q19 and D3Q27 lattice models have been included. The speed vectors and weight factors are arranged to allow the use of swap algorithms for propagation[34]. The transformation matrices $\mathbf{T}$ for Multiple-Relaxation-Time (MRT) schemes are included for each lattice (except D3Q27), along with the equilibrium moments ($\vec{M}^{eq}$) expressed for the incompressible case (for the compressible case, ρ_0 is substituted with ρ), collision operators ($\vec{s}$), definition of bulk viscosity (ν') and forcing source-terms ($\vec{S}$).

D2Q9

Weight factor

i	w_i
0	$\frac{4}{9}$
2,4,6,8	$\frac{1}{9}$
1,3,5,7	$\frac{1}{36}$

Speed vector

i	$e_{i,x}$	$e_{i,y}$
0	0	0
1	-1	1
2	-1	0
3	-1	-1
4	0	-1
5	1	-1
6	1	0
7	1	1
8	0	1

$$\mathbf{T} = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ -4 & 2 & -1 & 2 & -1 & 2 & -1 & 2 & -1 \\ 4 & 1 & -2 & 1 & -2 & 1 & -2 & 1 & -2 \\ 0 & -1 & -1 & -1 & 0 & 1 & 1 & 1 & 0 \\ 0 & -1 & 2 & -1 & 0 & 1 & -2 & 1 & 0 \\ 0 & 1 & 0 & -1 & -1 & -1 & 0 & 1 & 1 \\ 0 & 1 & 0 & -1 & 2 & -1 & 0 & 1 & -2 \\ 0 & 0 & 1 & 0 & -1 & 0 & 1 & 0 & -1 \\ 0 & -1 & 0 & 1 & 0 & -1 & 0 & 1 & 0 \end{bmatrix}$$

$$\vec{M}^{eq} = \begin{pmatrix} \rho \\ e^{eq} \\ \epsilon^{eq} \\ j_x \\ q_x^{eq} \\ j_y \\ q_y^{eq} \\ p_{xx}^{eq} \\ p_{xy}^{eq} \end{pmatrix} = \begin{pmatrix} \rho \\ -2\rho + \frac{3}{\rho_0}\left(j_x^2 + j_y^2\right) \\ w_\epsilon \rho + \frac{w_{\epsilon j}}{\rho_0}(j_x^2 + j_y^2) \\ j_x \\ -j_x \\ j_y \\ -j_y \\ \frac{j_x^2 - j_y^2}{3\rho_0} \\ \frac{j_x j_y}{3\rho_0} \end{pmatrix}$$

$$\vec{S} = \begin{pmatrix} 0 \\ 6(v_x F_x + v_y F_y) \\ -6(v_x F_x + v_y F_y) \\ F_x \\ -F_x \\ F_y \\ -F_y \\ 2(v_x F_x - v_y F_y) \\ v_x F_y + v_y F_x \end{pmatrix}.$$

$$\vec{s} = \left(1, \tau_{f,bulk}^{-1}, s_2, 1, s_4, 1, s_4, \tau_f^{-1}, \tau_f^{-1}\right)^T$$

$$\nu' = \frac{1}{6}\left(\tau_{f,bulk} - \frac{1}{2}\right)\frac{(\Delta x)^2}{\Delta t}$$

D3Q15

Weight factor

i	w_i
0	$\frac{2}{9}$
1–3, 8–10	$\frac{1}{9}$
4–7, 11–14	$\frac{1}{72}$

Speed vector

i	$e_{i,x}$	$e_{i,y}$	$e_{i,z}$
0	0	0	0
1	-1	0	0
2	0	-1	0
3	0	0	-1
4	-1	-1	-1
5	-1	-1	1
6	-1	1	-1
7	-1	1	1
8	1	0	0
9	0	1	0
10	0	0	1
11	1	1	1
12	1	1	-1
13	1	-1	1
14	1	-1	-1

$$\mathbf{T} = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ -2 & -1 & -1 & -1 & 1 & 1 & 1 & 1 & -1 & -1 & -1 & 1 & 1 & 1 & 1 \\ 16 & -4 & -4 & -4 & 1 & 1 & 1 & 1 & -4 & -4 & -4 & 1 & 1 & 1 & 1 \\ 0 & -1 & 0 & 0 & -1 & -1 & -1 & -1 & 1 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 4 & 0 & 0 & -1 & -1 & -1 & -1 & -4 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & -1 & 0 & -1 & -1 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & -1 & -1 \\ 0 & 0 & 4 & 0 & -1 & -1 & 1 & 1 & 0 & -4 & 0 & 1 & 1 & -1 & -1 \\ 0 & 0 & 0 & -1 & -1 & 1 & -1 & 1 & 0 & 0 & 1 & 1 & -1 & 1 & -1 \\ 0 & 0 & 0 & 4 & -1 & 1 & -1 & 1 & 0 & 0 & -4 & 1 & -1 & 1 & -1 \\ 0 & 2 & -1 & -1 & 0 & 0 & 0 & 0 & 2 & -1 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & -1 & 0 & 0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & -1 & -1 & 0 & 0 & 0 & 1 & 1 & -1 & -1 \\ 0 & 0 & 0 & 0 & 1 & -1 & -1 & 1 & 0 & 0 & 0 & 1 & -1 & -1 & 1 \\ 0 & 0 & 0 & 0 & 1 & -1 & 1 & -1 & 0 & 0 & 0 & 1 & -1 & 1 & -1 \\ 0 & 0 & 0 & 0 & -1 & 1 & 1 & -1 & 0 & 0 & 0 & 1 & -1 & -1 & 1 \end{bmatrix}$$

$$\vec{M}^{eq} = \begin{pmatrix} \rho \\ e^{eq} \\ \epsilon^{eq} \\ j_x \\ q_x^{eq} \\ j_y \\ q_y^{eq} \\ j_z \\ q_z^{eq} \\ 3p_{xx}^{eq} \\ p_{ww}^{eq} \\ p_{xy}^{eq} \\ p_{yz}^{eq} \\ p_{zx}^{eq} \\ m_{xyz}^{eq} \end{pmatrix} = \begin{pmatrix} \rho \\ -2\rho + \frac{\left(j_x^2+j_y^2+j_z^2\right)}{\rho_0} \\ w_\epsilon \rho + \frac{w_{\epsilon j}}{\rho_0}(j_x^2+j_y^2+j_z^2) \\ j_x \\ -\frac{7}{3}j_x \\ j_y \\ -\frac{7}{3}j_y \\ j_z \\ -\frac{7}{3}j_z \\ \frac{2j_x^2-j_y^2-j_z^2}{\rho_0} \\ \frac{j_y^2-j_z^2}{\rho_0} \\ \frac{j_x j_y}{\rho_0} \\ \frac{j_y j_z}{\rho_0} \\ \frac{j_z j_x}{\rho_0} \\ 0 \end{pmatrix}$$

$$\vec{S} = \begin{pmatrix} 0 \\ 2(v_xF_x + v_yF_y + v_zF_z) \\ -10(v_xF_x + v_yF_y + v_zF_z) \\ F_x \\ -\frac{7}{3}F_x \\ F_y \\ -\frac{7}{3}F_y \\ F_z \\ -\frac{7}{3}F_z \\ 2(2v_xF_x - v_yF_y - v_zF_z) \\ 2(v_yF_y - v_zF_z) \\ v_xF_y + v_yF_x \\ v_yF_z + v_zF_y \\ v_zF_x + v_xF_z \\ 0 \end{pmatrix}.$$

$$\vec{s} = \left(1, \tau_{f,bulk}^{-1}, s_2, 1, s_4, 1, s_4, 1, s_4, \tau_f^{-1}, \tau_f^{-1}, \tau_f^{-1}, \tau_f^{-1}, \tau_f^{-1}, s_{14}\right)^T$$

$$\nu' = \frac{2}{9}\left(\tau_{f,bulk} - \frac{1}{2}\right)\frac{(\Delta x)^2}{\Delta t}$$

D3Q19

Weight factor

i	w_i
0	$\frac{1}{3}$
1–3, 10–12	$\frac{1}{18}$
4–9, 13–18	$\frac{1}{36}$

Speed vector

i	$e_{i,x}$	$e_{i,y}$	$e_{i,z}$
0	0	0	0
1	-1	0	0
2	0	-1	0
3	0	0	-1
4	-1	-1	0
5	-1	1	0
6	-1	0	-1
7	-1	0	1
8	0	-1	-1
9	0	-1	1
10	1	0	0
11	0	1	0
12	0	0	1
13	1	1	0
14	1	-1	0
15	1	0	1
16	1	0	-1
17	0	1	1
18	0	1	-1

$$\mathbf{T} = \begin{bmatrix}
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
-30 & -11 & -11 & -11 & 8 & 8 & 8 & 8 & 8 & 8 & -11 & -11 & -11 & 8 & 8 & 8 & 8 & 8 & 8 \\
12 & -4 & -4 & -4 & 1 & 1 & 1 & 1 & 1 & 1 & -4 & -4 & -4 & 1 & 1 & 1 & 1 & 1 & 1 \\
0 & -1 & 0 & 0 & -1 & -1 & -1 & -1 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 \\
0 & 4 & 0 & 0 & -1 & -1 & -1 & -1 & 0 & 0 & -4 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 \\
0 & 0 & -1 & 0 & -1 & 1 & 0 & 0 & -1 & -1 & 0 & 1 & 0 & 1 & -1 & 0 & 0 & 1 & 1 \\
0 & 0 & 4 & 0 & -1 & 1 & 0 & 0 & -1 & -1 & 0 & -4 & 0 & 1 & -1 & 0 & 0 & 1 & 1 \\
0 & 0 & 0 & -1 & 0 & 0 & -1 & 1 & -1 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & -1 & 1 & -1 \\
0 & 0 & 0 & 4 & 0 & 0 & -1 & 1 & -1 & 1 & 0 & 0 & -4 & 0 & 0 & 1 & -1 & 1 & -1 \\
0 & 2 & -1 & -1 & 1 & 1 & 1 & 1 & -2 & -2 & 2 & -1 & -1 & 1 & 1 & 1 & 1 & -2 & -2 \\
0 & -4 & 2 & 2 & 1 & 1 & 1 & 1 & -2 & -2 & -4 & 2 & 2 & 1 & 1 & 1 & 1 & -2 & -2 \\
0 & 0 & 1 & -1 & 1 & 1 & -1 & -1 & 0 & 0 & 0 & 1 & -1 & 1 & 1 & -1 & -1 & 0 & 0 \\
0 & 0 & -2 & 2 & 1 & 1 & -1 & -1 & 0 & 0 & 0 & -2 & 2 & 1 & 1 & -1 & -1 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 \\
0 & 0 & 0 & 0 & -1 & -1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & -1 & -1 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 & -1 & -1 & 0 & 0 & 0 & -1 & 1 & 0 & 0 & 1 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & -1 & 1 & 1 & -1 & 0 & 0 & 0 & 0 & 0 & 1 & -1 & -1 & 1
\end{bmatrix}$$

$$\vec{M}^{eq} = \begin{pmatrix} \rho \\ e^{eq} \\ \epsilon^{eq} \\ j_x \\ q_x^{eq} \\ j_y \\ q_y^{eq} \\ j_z \\ q_z^{eq} \\ 3p_{xx}^{eq} \\ 3\pi_{xx}^{eq} \\ p_{ww}^{eq} \\ \pi_{ww}^{eq} \\ p_{xy}^{eq} \\ p_{yz}^{eq} \\ p_{zx}^{eq} \\ m_x^{eq} \\ m_y^{eq} \\ m_z^{eq} \end{pmatrix} = \begin{pmatrix} \rho \\ -11\rho + \frac{19}{\rho_0}\left(j_x^2 + j_y^2 + j_z^2\right) \\ w_\epsilon \rho + \frac{w_{ej}}{\rho_0}(j_x^2 + j_y^2 + j_z^2) \\ j_x \\ -\frac{2}{3} j_x \\ j_y \\ -\frac{2}{3} j_y \\ j_z \\ -\frac{2}{3} j_z \\ \frac{2j_x^2 - j_y^2 - j_z^2}{\rho_0} \\ \frac{w_{xx}}{\rho_0}\left(2j_x^2 - j_y^2 - j_z^2\right) \\ \frac{j_y^2 - j_z^2}{\rho_0} \\ \frac{w_{xx}}{\rho_0}\left(j_y^2 - j_z^2\right) \\ \frac{j_x j_y}{\rho_0} \\ \frac{j_y j_z}{\rho_0} \\ \frac{j_z j_x}{\rho_0} \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

$$\vec{S} = \begin{pmatrix} 0 \\ 38(v_x F_x + v_y F_y + v_z F_z) \\ -11(v_x F_x + v_y F_y + v_z F_z) \\ F_x \\ -\frac{2}{3} F_x \\ F_y \\ -\frac{2}{3} F_y \\ F_z \\ -\frac{2}{3} F_z \\ 2(2v_x F_x - v_y F_y - v_z F_z) \\ -(2v_x F_x - v_y F_y - v_z F_z) \\ 2(v_y F_y - v_z F_z) \\ -(v_y F_y - v_z F_z) \\ v_x F_y + v_y F_x \\ v_y F_z + v_z F_y \\ v_z F_x + v_x F_z \\ 0 \\ 0 \\ 0 \end{pmatrix}.$$

$$\vec{s} = \left(1, \tau_{f,bulk}^{-1}, s_2, 1, s_4, 1, s_4, 1, s_4, \tau_f^{-1}, s_4, \tau_f^{-1}, s_4, \tau_f^{-1}, \tau_f^{-1}, \tau_f^{-1}, s_{16}, s_{16}, s_{16}\right)^T$$

$$\nu' = \frac{2}{9}\left(\tau_{f,bulk} - \frac{1}{2}\right)\frac{(\Delta x)^2}{\Delta t}$$

D3Q27

Weight factor

i	w_i
0	$\frac{8}{27}$
1–3, 14–16	$\frac{2}{27}$
4–9, 17–22	$\frac{1}{54}$
10–13, 23–26	$\frac{1}{216}$

Speed vector

i	$e_{i,x}$	$e_{i,y}$	$e_{i,z}$
0	0	0	0
1	-1	0	0
2	0	-1	0
3	0	0	-1
4	-1	-1	0
5	-1	1	0
6	-1	0	-1
7	-1	0	1
8	0	-1	-1
9	0	-1	1
10	-1	-1	-1
11	-1	-1	1
12	-1	1	-1
13	-1	1	1
14	1	0	0
15	0	1	0
16	0	0	1
17	1	1	0
18	1	-1	0
19	1	0	1
20	1	0	-1
21	0	1	1
22	0	1	-1
23	1	1	1
24	1	1	-1
25	1	-1	1
26	1	-1	-1

Table 4.1: Boundary condition category

value	meaning
0	liquid
10	domain boundary
11	inside solid
12	on-grid bounce-back boundary
13	mid-link bounce-back boundary
100–199	constant speed, composition and temperature boundary
200–299	constant speed, Neumann composition and temperature boundary
300–399	constant speed and composition, Neumann temperature boundary
400–499	constant speed and temperature, Neumann composition boundary
500–599	constant pressure, composition and temperature boundary
600–699	constant pressure, Neumann composition and temperature boundary
700–799	constant pressure and composition, Neumann temperature boundary
800–899	constant pressure and temperature, Neumann composition boundary

4.2 Data structure

4.2.1 Storage of particle distribution functions

For a system with a square lattice, the total number of grid points = `lbsy.nx` × `lbsy.ny` × `lbsy.nz`, where `lbsy.nx`, `lbsy.ny` and `lbsy.nz` are the numbers of grid points along the x-, y- and z-axes respectively. The grid points are arranged in a serial order of

$$g_{000}, g_{001}, \cdots g_{00\,\texttt{lbsy.nz}}, g_{010}, g_{011}, \cdots g_{0\,\texttt{lbsy.ny}\,\texttt{lbsy.nz}}, g_{100}, g_{101}, \cdots g_{\texttt{lbsy.nx}\,\texttt{lbsy.ny}\,\texttt{lbsy.nz}}$$

At each grid point, DL_MESO_LBE arranges the particle distribution functions in order of: fluid functions, solute functions, temperature functions and phase field order parameter. For example, for a D2Q9 lattice with two fluids, scalar temperature and phase field, the distribution functions are in the order of

$$f_0^0, f_1^0, f_2^0, f_3^0, f_4^0, f_5^0, f_6^0, f_7^0, f_8^0, f_0^1, f_1^1, f_2^1, f_3^1, f_4^1, f_5^1, f_6^1, f_7^1, f_8^1, T_0, T_1, T_2, T_3, T_4, T_5, T_6, T_7, T_8, pf$$

Therefore the number of particle distribution functions at each grid point is

$$\texttt{lbsitelength} = (\texttt{lbsy.nf} + \texttt{lbsy.nc} + \texttt{lbsy.nt}) \times \texttt{lbsy.nq} + \texttt{lbsy.ph}$$

where `lbsy.nf`, `lbsy.nc`, `lbsy.nt`, `lbsy.nq` and `lbsy.ph` are respectively: number of fluids, number of solutes, number of temperature scalars, number of discrete speeds and number of the phase field order parameters. `lbsy.nt` and `lbsy.ph` can only take the values of 1 or 0, representing systems with or without temperature scalar and phase field. Also if `lbsy.nc` $\neq 0$, `lbsy.nf` cannot be set larger than 1.

4.2.2 Storage of space property

The space property is represented by an integer value in DL_MESO_LBE. For example, `lbphi`[100] = 0 represents the 100th grid point as a liquid site and `lbphi`[101] = 12 shows the 101st grid point as an on-grid bounce-back boundary. Table 4.1 lists the categories of space properties.

The orientation of a solid-liquid boundary is also represented by the value of an integer. For example, a planar surface with normal vector along the y-axis is denoted by 21, while a concave corner face at the top-right-front corner is denoted by 31. It must be pointed out that only those space positions located in the surface of a face-centered cube have been included and translated in DL_MESO_LBE. Points with random orientations, e.g. 47° plane, have not been included.

The boundary condition number can be rather confusing and difficult to understand. The GUI in DL_MESO therefore includes a translator which interprets a *word* as its corresponding integer number. The *word* is made

Table 4.2: Boundary condition category

letter	**meaning**
V	Constant Velocity
P	Constant Pressure (Density)
C	Constant Solute Composition
T	Constant Temperature
B	Neumann Boundary Condition (Solute Composition or Temperature)
PS	Planar Surface
CC	Concave Corner
CE	Concave Edge
T	Normal Vector Pointing to Top
D	Normal Vector Pointing Downwards
L	Normal Vector Pointing to Left
R	Normal Vector Pointing to Right
F	Normal Vector Pointing to Front
B	Normal Vector Pointing to Back

up of defined letters as listed in Table 4.2. The boundary conditions with combinations of type and orientation are listed in Table 4.3. The *letters* are in the order of:

1. Fluid property: constant speed or constant pressure.
2. Solute property: constant composition or Neumann boundary.
3. Temperature property: isothermal (constant) or heat bath (Neumann boundary).
4. Geometric property: planar surface, concave corner or concave edge.
5. Boundary orientation: one letter for planar surface, two letters for concave corners or three letters for concave edges.

For example, a shearing planar surface facing down the y-axis with constant composition and temperature (i.e. isothermal) is represented as VCBPSD and translated as 322.

Table 4.3: Notation of boundary condition

VCTPST	121	VCTPSD	122	VCTPSL	123
VCTPSR	124	VCTPSF	125	VCTPSB	126
VCTCCTRB	127	VCTCCTLB	128	VCTCCDLB	129
VCTCCDRB	130	VCTCCTRF	131	VCTCCTLF	132
VCTCCDLF	133	VCTCCDRF	134	VCTCETR	143
VCTCETL	144	VCTCEDL	145	VCTCEDR	146
VCTCETF	147	VCTCELF	148	VCTCEDF	149
VCTCERF	150	VCTCETB	151	VCTCELB	152
VCTCEDB	153	VCTCERB	154	VBBPST	221
VBBPSD	222	VBBPSL	223	VBBPSR	224
VBBPSF	225	VBBPSB	226	VBBCCTRB	227
VBBCCTLB	228	VBBCCDLB	229	VBBCCDRB	230
VBBCCTRF	231	VBBCCTLF	232	VBBCCDLF	233
VBBCCDRF	234	VBBCETR	243	VBBCETL	244
VBBCEDL	245	VBBCEDR	246	VBBCETF	247
VBBCELF	248	VBBCEDF	249	VBBCERF	250
VBBCETB	251	VBBCELB	252	VBBCEDB	253
VBBCERB	254	VCBPST	321	VCBPSD	322
VCBPSL	323	VCBPSR	324	VCBPSF	325
VCBPSB	326	VCBCCTRB	327	VCBCCTLB	328
VCBCCDLB	329	VCBCCDRB	330	VCBCCTRF	331
VCBCCTLF	332	VCBCCDLF	333	VCBCCDRF	334
VCBCETR	343	VCBCETL	344	VCBCEDL	345
VCBCEDR	346	VCBCETF	347	VCBCELF	348
VCBCEDF	349	VCBCERF	350	VCBCETB	351
VCBCELB	352	VCBCEDB	353	VCBCERB	354
VBTPST	421	VBTPSD	422	VBTPSL	423
VBTPSR	424	VBTPSF	425	VBTPSB	426
VBTCCTRB	427	VBTCCTLB	428	VBTCCDLB	429
VBTCCDRB	430	VBTCCTRF	431	VBTCCTLF	432
VBTCCDLF	433	VBTCCDRF	434	VBTCETR	443
VBTCETL	444	VBTCEDL	445	VBTCEDR	446
VBTCETF	447	VBTCELF	448	VBTCEDF	449
VBTCERF	450	VBTCETB	451	VBTCELB	452
VBTCEDB	453	VBTCERB	454	PCTPST	521
PCTPSD	522	PCTPSL	523	PCTPSR	524
PCTPSF	525	PCTPSB	526	PCTCCTRB	527
PCTCCTLB	528	PCTCCDLB	529	PCTCCDRB	530
PCTCCTRF	531	PCTCCTLF	532	PCTCCDLF	533
PCTCCDRF	534	PCTCETR	543	PCTCETL	544
PCTCEDL	545	PCTCEDR	546	PCTCETF	547
PCTCELF	548	PCTCEDF	549	PCTCERF	550
PCTCETB	551	PCTCELB	552	PCTCEDB	553
PCTCERB	554	PBTPST	621	PBTPSD	622
PBTPSL	623	PBTPSR	624	PBTPSF	625
PBTPSB	626	PBTCCTRB	627	PBTCCTLB	628
PBTCCDLB	629	PBTCCDRB	630	PBTCCTRF	631
PBTCCTLF	632	PBTCCDLF	633	PBTCCDRF	634

Table 4.3: Notation of boundary condition (continued)

PBTCETR	643	PBTCETL	644	PBTCEDL	645
PBTCEDR	646	PBTCETF	647	PBTCELF	648
PBTCEDF	649	PBTCERF	650	PBTCETB	651
PBTCELB	652	PBTCEDB	653	PBTCERB	654
PCBPST	721	PCBPSD	722	PCBPSL	723
PCBPSR	724	PCBPSF	725	PCBPSB	726
PCBCCTRB	727	PCBCCTLB	728	PCBCCDLB	729
PCBCCDRB	730	PCBCCTRF	731	PCBCCTLF	732
PCBCCDLF	733	PCBCCDRF	734	PCBCETR	743
PCBCETL	744	PCBCEDL	745	PCBCEDR	746
PCBCETF	747	PCBCELF	748	PCBCEDF	749
PCBCERF	750	PCBCETB	751	PCBCELB	752
PCBCEDB	753	PCBCERB	754	PBBPST	821
PBBPSD	822	PBBPSL	823	PBBPSR	824
PBBPSF	825	PBBPSB	826	PBBCCTRB	827
PBBCCTLB	828	PBBCCDLB	829	PBBCCDRB	830
PBBCCTRF	831	PBBCCTLF	832	PBBCCDLF	833
PBBCCDRF	834	PBBCETR	843	PBBCETL	844
PBBCEDL	845	PBBCEDR	846	PBBCETF	847
PBBCELF	848	PBBCEDF	849	PBBCERF	850
PBBCETB	851	PBBCELB	852	PBBCEDB	853
PPBCERB	854				

4.2.3 Storage of running information

The Lattice Boltzmann component of DL_MESO defines three structures to store the system[1] information. The parameters in these structures are listed in Tables 4.4, 4.5 and 4.6.

Table 4.4: System information

parameter	**meaning**
`lbsy.nd`	space dimension
`lbsy.nq`	number of discrete speeds
`lbsy.nf`	number of fluids
`lbsy.nc`	number of solutes
`lbsy.nt`	number of temperature scalars (0 or 1)
`lbsy.pf`	phase field order parameter
`lbsy.nx`	number of grid points in x direction
`lbsy.ny`	number of grid points in y direction
`lbsy.nz`	number of grid points in z direction (`lbsy.nz` $\equiv 1$ when `lbsy.nd` $= 2$)

[1]Referring to the physical system being simulated rather than the computer system.

Table 4.5: Domain information

parameter	meaning
lbdm.rank	name of the processor
lbdm.size	number of processors
lbdm.bwid	domain boundary width (set to zero for serial running)
lbdm.xcor	x-coordinate of the processor
lbdm.ycor	y-coordinate of the processor
lbdm.zcor	z-coordinate of the processor (lbdm.zcor $\equiv 0$ when lbsy.nd $= 2$)
lbdm.xdim	number of processors along x-axis
lbdm.ydim	number of processors along y-axis
lbdm.zdim	number of processors along z-axis (lbdm.zdim $\equiv 1$ when lbsy.nd $= 2$)
lbdm.xs	x-coordinate of domain start position
lbdm.xe	x-coordinate of domain end position
lbdm.xinner	number of grid points along x-axis in the domain
lbdm.xouter	number of grid points along x-axis in the domain including the boundary
lbdm.ys	y-coordinate of domain start position
lbdm.ye	y-coordinate of domain end position
lbdm.yinner	number of grid points along y-axis in the domain
lbdm.youter	number of grid points along y-axis in the domain including the boundary
lbdm.zs	z-coordinate of domain start position
lbdm.ze	z-coordinate of domain end position
lbdm.zinner	number of grid points along z-axis in the domain
lbdm.zouter	number of grid points along z-axis in the domain including the boundary
lbdm.touter	total number of grid points in the domain including the boundary

Table 4.6: Neighbour information

parameter	meaning
lbnb[k].rank	processor name of neighbour k
lbnb[k].spos	start position for sending distribution function message
lbnb[k].rpos	start position for receiving distribution function message
lbnb[k].bspos	start position for sending boundary condition message
lbnb[k].brpos	start position for receiving boundary condition message
k = 0	right neighbour
k = 1	left neighbour
k = 2	upper neighbour
k = 3	lower neighbour
k = 4	front neighbour
k = 5	back neighbour

4.3 The Parameters and Their Functions

Table 4.7 lists all the parameters defined in DL_MESO_LBE. The whole range parameters are named with the prefix lb. Because DL_MESO is an ongoing project and new parameters might be added to the package in the future, it is strongly suggested that users of DL_MESO would not name their own parameters with prefixes lb, dp or sp.

The notation column in Table 4.7 gives the restrictions applicable on the parameters. 'A' indicates an array of data, followed by the number of elements in the array. For example, 'A lbsy.nf' means the parameter is actually an array with lbsy.nf elements. '≥ 1' means the number must be greater or equal to one, while '1 or 0' means the value of the parameter can either be one or zero. An asterisk in the data type for the array indicates that it is allocatable.

Table 4.7: DL_MESO_LBE Parameters

function	**parameter**	**data type**	**notation**
system information	`lbsy`	structure	
domain information	`lbdm`	structure	
neighbour information	`lbnb`	structure	A 6
space dimension	`lbsy.nd`	int	
number of discrete speeds	`lbsy.nq`	int	
number of fluids	`lbsy.nf`	int	≥ 1
number of solutes	`lbsy.nc`	int	
temperature scalars	`lbsy.nt`	int	1 or 0
grid points in x-direction	`lbsy.nx`	int	
grid points in y-direction	`lbsy.ny`	int	
grid points in z-direction	`lbsy.nz`	int	
domain boundary width	`lbsy.bwid`	int	≥ 1
system dimension along x	`lbxsize`	double	
total calculation steps	`lbtotstep`	int	
data save interval	`lbsave`	int	
steering	`lbsteer`	int	1 or 0
system molar weight	`lbsmw`	double	
noise intensity	`lbnoise`	double	
system molar volume	`lbsmv`	double	
initial system velocity	`lbiniv`	double	A 3
top boundary velocity	`lbtopv`	double	A 3
bottom boundary velocity	`lbbotv`	double	A 3
front boundary velocity	`lbfrov`	double	A 3
back boundary velocity	`lbbakv`	double	A 3
left boundary velocity	`lblefv`	double	A 3
right boundary velocity	`lbrigv`	double	A 3
constant incompressible fluid density (ρ_0)	`lbincp`	double*	A `lbsy.nf`
initial fluid density	`lbinip`	double*	A `lbsy.nf`
top boundary fluid density	`lbtopp`	double*	A `lbsy.nf`
bottom boundary fluid density	`lbbotp`	double*	A `lbsy.nf`
front boundary fluid density	`lbfrop`	double*	A `lbsy.nf`
back boundary fluid density	`lbbakp`	double*	A `lbsy.nf`
left boundary fluid density	`lblefp`	double*	A `lbsy.nf`
right boundary fluid density	`lbrigp`	double*	A `lbsy.nf`
initial composition	`lbinic`	double*	A `lbsy.nc`
top boundary composition	`lbtopc`	double*	A `lbsy.nc`
bottom boundary composition	`lbbotc`	double*	A `lbsy.nc`
front boundary composition	`lbfroc`	double*	A `lbsy.nc`
back boundary composition	`lbbakc`	double*	A `lbsy.nc`
left boundary composition	`lblefc`	double*	A `lbsy.nc`
right boundary composition	`lbrigc`	double*	A `lbsy.nc`
initial temperature	`lbinit`	double	
top boundary temperature	`lbtopt`	double	
bottom boundary temperature	`lbbott`	double	
front boundary temperature	`lbfrot`	double	
back boundary temperature	`lbbakt`	double	
left boundary temperature	`lbleft`	double	
right boundary temperature	`lbrigt`	double	
system cooling rate	`lbsysdt`	double	
top boundary cooling rate	`lbtopdt`	double	
bottom boundary cooling rate	`lbbotdt`	double	
front boundary cooling rate	`lbfrodt`	double	

Table 4.7: DL_MESO_LBE Parameters (continued)

function	parameter	data type	notation
back boundary cooling rate	`lbbakdt`	double	
left boundary cooling rate	`lblefdt`	double	
right boundary cooling rate	`lbrigdt`	double	
fluid inverse relaxation time (τ_f^{-1})	`lbtf`	double*	A `lbsy.nf`
bulk fluid inverse relaxation time ($\tau_{f,bulk}^{-1}$)	`lbtfbulk`	double*	A `lbsy.nf`
solute inverse relaxation time (τ_c^{-1})	`lbtc`	double*	A `lbsy.nc`
temperature inverse relaxation time (τ_t^{-1})	`lbtt`	double*	A `lbsy.nt`
particle interaction function (g_{ab})	`lbg`	double*	A `lbsy.nf*lbsy.nf`
body force	`lbbdforce`	double*	A `3*lbsy.nf`
interaction force	`lbinterforce`	double*	A `3*lbsy.nf*lbdm.touter`
distribution function	`lbf`	double*	A `lbsitelength*lbdm.touter`
temporary function	`lbft`	double*	A `lbdm.touter*(lbsy.nf+lbsy.nc)`
equilibrium distribution	`lbfeq`	double*	A `lbsy.nq`
speed vector for the model	`lbv`	int*	A `3*lbsy.nq`
index for opposing speed	`lbopv`	int*	A `lbsy.nq`
weight factor of speed vector	`lbw`	double*	A `lbsy.nq`
MRT transformation matrix	`lbtr`	double*	A `lbsy.nq*lbsy.nq`
MRT inverse transformation matrix	`lbtrinv`	double*	A `lbsy.nq*lbsy.nq`
MRT tuneable collision parameters	`lbmrts`	double	A 3
MRT tuneable equilibrium parameters	`lbmrtw`	double	A 3
number of parameters per grid point	`lbsitelength`	int	
number of grid points in yz plane	`lbyz`	int	
grid spacing	`lbdx`	double	
time step	`lbdt`	double	
speed of sound	`lbsoundv`	double	
Reynolds number	`lbreynolds`	double	
processor name	`lbdm.rank`	int	
total number of processors	`lbdm.size`	int	
x-coordinate for domain	`lbdm.xcor`	int	
y-coordinate for domain	`lbdm.ycor`	int	
z-coordinate for domain	`lbdm.zcor`	int	
number of processors along x-axis	`lbdm.xdim`	int	
number of processors along y-axis	`lbdm.ydim`	int	
number of processors along z-axis	`lbdm.zdim`	int	
x-coordinate of domain start position	`lbdm.xs`	int	
x-coordinate of domain end position	`lbdm.xe`	int	
y-coordinate of domain start position	`lbdm.ys`	int	
y-coordinate of domain end position	`lbdm.ye`	int	
z-coordinate of domain start position	`lbdm.zs`	int	
z-coordinate of domain end position	`lbdm.ze`	int	
inner[2] grid points along x	`lbdm.xinner`	int	
outer[3] grid points along x	`lbdm.xouter`	int	
inner grid points along y	`lbdm.yinner`	int	
outer grid points along y	`lbdm.youter`	int	
inner grid points along z	`lbdm.zinner`	int	
outer grid points along z	`lbdm.zouter`	int	
total grid points	`lbdm.touter`	int	
name of neighbouring processor	`lbnb[].rank`	int	
position for sending distribution message	`lbnb[].spos`	unsigned long	
position for receiving distribution message	`lbnb[].rpos`	unsigned long	

[2]Excluding boundary points
[3]Including boundary points

Table 4.7: DL_MESO_LBE Parameters (continued)

function	parameter	data type	notation
position for sending boundary message	`lbnb[].bspos`	unsigned long	
position for receiving boundary message	`lbnb[].brpos`	unsigned long	
position for sending force message	`lbnb[].fspos`	unsigned long	
position for receiving force message	`lbnb[].frpos`	unsigned long	
message type	`lbmsg2x`	MPI_Datatype	
message type	`lbmsg2y`	MPI_Datatype	
message type	`lbmsg3x`	MPI_Datatype	
message type	`lbmsg3y`	MPI_Datatype	
message type	`lbmsg3z`	MPI_Datatype	
message type	`lbbmsg2x`	MPI_Datatype	
message type	`lbbmsg2y`	MPI_Datatype	
message type	`lbbmsg3x`	MPI_Datatype	
message type	`lbbmsg3y`	MPI_Datatype	
message type	`lbbmsg3z`	MPI_Datatype	
message type	`lbfmsg2x`	MPI_Datatype	
message type	`lbfmsg2y`	MPI_Datatype	
message type	`lbfmsg3x`	MPI_Datatype	
message type	`lbfmsg3y`	MPI_Datatype	
message type	`lbfmsg3z`	MPI_Datatype	
endianness of system	`bigend`	int	
total calculation time	`totaltime`	double	
output file number	`qVersion`	int	

Chapter 5

DL_MESO_LBE Features

5.1 Collision and Propagation Algorithms

The collision and propagation routines are a fundamental part of Lattice Boltzmann Equation calculations. Implementation involves the calculation of post-collisional values for the distribution functions at each lattice point (at time t^+). For the generalized form of the Lattice Boltzmann Equation with the collision operator C_i (normally in the form of a matrix):

$$f_i\left(\vec{x}, t^+\right) = f_i\left(\vec{x}, t\right) + C_i$$

and movement of these distribution functions to neighbouring lattice nodes:

$$f_i\left(\vec{x} + \hat{e}_i \Delta t, t + \Delta t\right) = f_i\left(\vec{x}, t^+\right)$$

which combine to give the governing equation for calculations.

5.1.1 Collision algorithms

The forms of collision currently available in DL_MESO_LBE are the Bhatnagar-Gross-Krook (BGK) single relaxation time[4] and Multiple Relaxation Time (MRT) schemes.

5.1.1.1 BGK single relaxation time

The BGK collision operator is defined by

$$C_i = -\frac{\Delta t}{\tau_f}\left(f_i\left(\vec{x}, t\right) - f_i^{eq}\right) \tag{5.1}$$

where τ_f is the relaxation time, related to the kinetic (shear) viscosity of fluid by

$$\nu = \frac{1}{3}\left(\tau_f - \frac{1}{2}\right)\frac{\left(\Delta x\right)^2}{\Delta t}$$

with the kinematic bulk viscosity of the fluid, ν', set equal to $\frac{2}{3}\nu$[6]. This reduces the equation for post-collisional distribution functions to

$$f_i\left(\vec{x}, t^+\right) = f_i\left(\vec{x}, t\right) - \frac{\Delta t}{\tau_f}\left(f_i\left(\vec{x}, t\right) - f_i^{eq}\right),$$

the same as Equation 3.12.

Interfacial and external forces are dealt with either by adding $\frac{\tau_f \vec{F}}{\rho}$ to the velocity of the fluid[32] when calculating the equilibrium distribution function f_i^{eq}, or by adding a forcing term to the collisional distribution function[14]:

$$F_i = \left(1 - \frac{1}{2\tau_f}\right) w_i \left[\frac{\hat{e}_i - \vec{v}}{c_s^2} + \frac{\left(\hat{e}_i \cdot \vec{v}\right)}{c_s^4}\hat{e}_i\right] \cdot \vec{F} \tag{5.2}$$

where $\vec{v}$ is defined as equal to $\vec{u} + \frac{\Delta t}{2\rho}\vec{F}$ and used in both the expression above and as the velocity for calculating equilibrium distribution functions. The former is used by default in `fCollisionBGK`, while the latter method by Guo *et al.* can be invoked using `fCollisionBGKGuo`.

5.1.1.2 Multiple Relaxation Time (MRT)

The MRT collision operator operates on a similar principle to the quasilinear Lattice Boltzmann Equation[18], which expresses collisions in terms of a square matrix with dimensions equal to the number of lattice links per grid point ($m \equiv$ `lbsy.nq`). Unlike quasilinear LBE, however, MRT collision schemes are applied to the moments for each lattice point rather than the distribution functions[26, 7], which are related to each other by

$$\vec{M} = \mathbf{T}\vec{f} \tag{5.3}$$

where $\vec{f}$ is a vector of all m distribution functions for the point, i.e. $(f_0, f_1 \dots f_{m-1})^T$, $\vec{M}$ the vector of moments (also of size m and dependent on the lattice system) and $\mathbf{T}$ the transformation matrix that renders the moments in terms of the distribution functions. Equilibrium values for the moments, $\vec{M}^{eq}$, can be determined by transforming the standard local equilibrium distribution function into moment space by

$$\vec{M}^{eq} = \mathbf{T}\vec{f}^{eq} \tag{5.4}$$

where $\vec{f}^{eq}$ is the vector of local equilibrium distribution functions: the resulting equilibrium moments can alternatively be expressed directly as functions of fluid density and velocity. Certain moments, such as density and momentum, must be conserved and their equilibrium values are set so that no changes are made. The post-collisional moments are determined by relaxation of the non-equilibrium part, i.e.

$$\vec{M}(\vec{x}, t^+) = \vec{M}(\vec{x}, t) - \mathbf{\Lambda}(\vec{M}(\vec{x}, t) - \vec{M}^{eq}(\vec{x}, t)) \tag{5.5}$$

where $\mathbf{\Lambda}$ is the collision matrix, which takes the form of a diagonal matrix of m collision parameters (represented by $\vec{s}$):

$$\mathbf{\Lambda} = diag(\vec{s}) \tag{5.6}$$

Some of the collision parameters can be specified by the user to set both kinematic and bulk viscosities, a few others can be tuned to improve simulation stability and the remainder (i.e. those for density and momentum) are fixed to conserve macroscopic hydrodynamics. If all values of s_i are set to $\frac{1}{\tau_f}$, the scheme reduces to BGK single relaxation time collisions. Since the collisional matrix is diagonal, equation 5.5 can be rewritten in terms of each moment, i.e.

$$M_i(\vec{x}, t^+) = M_i(\vec{x}, t) - s_i \left(M_i(\vec{x}, t) - M_i^{eq}(\vec{x}, t)\right) \tag{5.7}$$

Multiplying $\vec{M}(\vec{x}, t^+)$ by the inverse of the transformation matrix, $\mathbf{T}^{-1}$, gives the post-collisional distribution functions.

Interfacial and external forces can be applied either by the addition of $\tau_f \vec{F}$ to the fluid momentum[32] or by the use of a moment-transformed source term, $\vec{S}$, whose terms are defined by[39]:

$$S_i = w_i \left[\frac{\hat{e}_i - \vec{v}}{c_s^2} + \frac{(\hat{e}_i \cdot \vec{v})}{c_s^4}\hat{e}_i\right] \cdot \vec{F} \tag{5.8}$$

and applied by the following to correct the post-collisional moments:

$$\Delta\vec{M} = \left(\mathbf{I} - \tfrac{1}{2}\mathbf{\Lambda}\right)\vec{S}\Delta t \tag{5.9}$$

where $\mathbf{I}$ as an identity matrix. The above equation can be re-expressed as

$$\Delta M_i = \left(1 - \tfrac{1}{2}s_i\right) S_i \Delta t \tag{5.10}$$

The above equations after inverse transformation reduce to equation 5.2 when the collision parameters are set to $\frac{1}{\tau_f}$. All collision parameters for conserved moments should be set to unity when applying external forces.

An example can be given for the D2Q9 lattice system[26]; the moment vector is

$$\vec{M} = (\rho, e, \epsilon, j_x, q_x, j_y, q_y, p_{xx}, p_{xy})^T$$

with ρ as the density, e the energy, ϵ the square of energy, $\vec{j}$ momentum, $\vec{q}$ energy flux, p_{xx} the diagonal stress tensor component and p_{xy} the off-diagonal stress tensor component. The transformation matrix is

$$\mathbf{T} = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ -4 & 2 & -1 & 2 & -1 & 2 & -1 & 2 & -1 \\ 4 & 1 & -2 & 1 & -2 & 1 & -2 & 1 & -2 \\ 0 & -1 & -1 & -1 & 0 & 1 & 1 & 1 & 0 \\ 0 & -1 & 2 & -1 & 0 & 1 & -2 & 1 & 0 \\ 0 & 1 & 0 & -1 & -1 & -1 & 0 & 1 & 1 \\ 0 & 1 & 0 & -1 & 2 & -1 & 0 & 1 & -2 \\ 0 & 0 & 1 & 0 & -1 & 0 & 1 & 0 & -1 \\ 0 & -1 & 0 & 1 & 0 & -1 & 0 & 1 & 0 \end{bmatrix}$$

The equilibrium moment vector is

$$\vec{M}^{eq} = \left(\rho, -2\rho + \frac{3(j_x^2 + j_y^2)}{\rho}, w_\epsilon \rho + w_{\epsilon j}\frac{(j_x^2 + j_y^2)}{\rho}, j_x, -j_x, j_y, -j_y, \frac{j_x^2 - j_y^2}{3\rho}, \frac{j_x j_y}{3\rho}\right)^T$$

with w_ϵ and $w_{\epsilon j}$ as adjustable parameters: for convergence to the single relaxation time BGK scheme, these are set equal to 1 and -3 respectively. Of the 9 collision parameters available, s_0, s_3 and s_5 have no effect (except when applying external forces, when they should be set equal to one) as the associated moments are preserved and s_2, s_4 and s_6 are tuneable parameters for calculational stability with the condition that $s_4 = s_6$. The remaining parameters represent the viscosities of the fluid, i.e.

$$\begin{aligned} \nu &= \frac{1}{3}\left(\frac{1}{s_7} - \frac{1}{2}\right)\frac{(\Delta x)^2}{\Delta t} = \frac{1}{3}\left(\frac{1}{s_8} - \frac{1}{2}\right)\frac{(\Delta x)^2}{\Delta t} \equiv \frac{1}{3}\left(\tau_f - \frac{1}{2}\right)\frac{(\Delta x)^2}{\Delta t} \\ \nu' &= \frac{1}{6}\left(\frac{1}{s_1} - \frac{1}{2}\right)\frac{(\Delta x)^2}{\Delta t} \equiv \frac{1}{6}\left(\tau_{f,bulk} - \frac{1}{2}\right)\frac{(\Delta x)^2}{\Delta t} \end{aligned}$$

i.e. $\tau_f = \frac{1}{s_7} = \frac{1}{s_8}$ and $\tau_{f,bulk} = \frac{1}{s_1}$. If the moment-transformed source terms are to be used, the vector $\vec{S}$ for this lattice scheme is defined as

$$\vec{S} = \begin{pmatrix} 0 \\ 6(u_x F_x + u_y F_y) \\ -6(u_x F_x + u_y F_y) \\ F_x \\ -F_x \\ F_y \\ -F_y \\ 2(u_x F_x - u_y F_y) \\ u_x F_y + u_y F_x \end{pmatrix}.$$

Similar schemes are available for the D3Q15 and D3Q19 lattices (there is currently no MRT scheme available for D3Q27). The MRT schemes without source terms can be applied using `fCollisionMRT`, while the schemes with Guo-like moment-transformed source terms can be invoked using `fCollisionMRTGuo`.

5.1.2 Propagation algorithms

The simplest implementation, the *two-lattice algorithm*, involves the use of a temporary array (`lbft`) to copy post-collisional distribution functions to their new positions, which are subsequently copied back to the main distribution function array `lbf`. While this particular method is clear, easy to understand and can be applied throughout the system's lattice points in any order, its drawbacks include the use of two loops for propagation

and array copying, two large arrays for the distribution functions at each lattice node and significant amounts of time expended in memory access. If the user wishes to use this method, the routine `fPropagation` is available.

An alternative, more efficient implementation of propagation is the *swap algorithm*[34], whereby this process is carried out by systematic swapping of pairs of collided distribution function values. To make this easier to implement, the lattice links are organised so that the conjugate link j to link i (i.e. $\hat{e}_j = -\hat{e}_i$) is equal to $i + \frac{m-1}{2}$ for $i = 1 \ldots \frac{m-1}{2}$. Looping i between 1 and $\frac{m-1}{2}$ the post-collisional distribution functions for each lattice point $f_i(\vec{x})$ are initially swapped with their conjugate values $f_j(\vec{x})$. then at each point the value $f_j(\vec{x})$ is then swapped with $f_i(\vec{x} + \hat{e}_i \Delta t)$.

These sets of swaps can be carried out either in two separate steps or in one go. The use of separate swap steps requires two sweeps through the domain, but the order in which the distribution functions are swapped does not matter and no boundary domain is necessary for serial calculations. Simultaneous swapping cannot make use of automatic periodic boundary conditions (thus a non-zero domain boundary is required) and requires lattice links to be additionally ordered so that the first half are all directed to lattice points that have previously gone through at least the first swap stage, but only a single sweep through the domain is required.

By default the serial version of DL_MESO_LBE uses the propagation routine that carries out two separate swap steps – `fPropagationSwap` – while the parallel version uses the combined-swap propagation routine `fPropagationCombSwap`. If the user wishes to use the combined-swap propagation routine in serial, an alternative program `slbecombine.cpp` is available which includes the following modifications to `slbe.cpp`:

- Replacement of `fSetSerialDomain` with `fSetSerialDomainBuffer`.
- Addition of `fsBoundPeriodic` and `fMarkBoundArea` immediately before `fInitializeSystem`.
- Addition of `fsPeriodic` inside main loop before calculating interaction forces (it does not matter whether this is placed before or after the call for creating output files).
- Addition of `fsForcePeriodic` inside main loop after calculating interaction forces (if these are not to be calculated, this call can be omitted).

5.2 Boundary conditions

To apply boundary conditions to a Lattice Boltzmann Equation simulation, the distribution functions f_i at boundary lattice points have to be modified or replaced during each time step to give the required fluid velocity or pressure/concentration/temperature. This may take place either between the collision and propagation stages or at the end of each time step: the subroutines `fPostCollBoundary` and `fPostPropBoundary` respectively are used to invoke the boundary conditions. The space property `lbphi` is used to define the boundary conditions for each lattice node in the system.

5.2.1 Periodic

Periodic boundaries are used to simulate bulk fluids sufficiently far away from the actual boundaries of a real physical system so that surface effects can be neglected. As the fluid moves out of one face of the system volume, it reappears on the opposite face with the same velocity, density etc.

DL_MESO_LBE applies periodic boundary conditions in two different ways depending on the size of the boundary domain `lbdm.bwid`. If there is no boundary domain (the default for serial running), periodic boundary conditions are automatically applied during the propagation step by using the function `fCppMod` to adjust the destination of distribution functions leaving the system so they are placed at the opposite side. No periodic boundary using this implementation needs to be defined by the space property, which can be left equal to zero as for the bulk fluid.

For systems that include a non-zero boundary domain size, the distribution functions at the edges of the actual system are copied into the buffer at the opposing sides before collisions and propagation take place. This requires the use of the space property (`lbphi[i]=10`) to determine the location of the domain buffer – which can be set up using the routine `fMarkBoundArea` – and either `fsPeriodic` and similar routines for serial running or `fNonBlockCommunication` and similar routines for running in parallel to copy the distribution functions into the buffers.

5.2.2 On-grid bounce-back

The on-grid bounce-back condition applies a no-slip condition (i.e. zero fluid velocity) at a boundary that lies halfway between grid points. This is applied after the propagation stage by reversing the distribution functions sitting on each wall node ($\vec{x}_w$), i.e.

$$f_i(\vec{x}_w, t) = f_j(\vec{x}_w, t) \tag{5.11}$$

where j is the opposite lattice link to i, i.e. $\hat{e}_j = -\hat{e}_i$. The reflection of distribution functions occurs *on-grid*. On-grid bounce-back is a first-order approximation of the boundary condition[36], i.e. the error is proportional to the lattice spacing Δx, but it is completely local (i.e. only uses distribution functions at the wall node).

5.2.3 Mid-grid bounce-back

Like the on-grid version, the mid-grid bounce-back condition applies a no-slip condition at a boundary halfway between lattice points[50]. This is applied by assigning post-collisional distribution functions to the wall node based on those values at neighbouring points, i.e.

$$f_i(\vec{x}_w, t^+) = f_j(\vec{x}_w + \hat{e}_i \Delta t, t^+). \tag{5.12}$$

This method essentially applies the actual reflection halfway between timesteps and is a spatially second-order method, although it is weakly non-local due to its use of distribution functions from neighbouring nodes.

5.2.4 Constant pressure/velocity

To specify either velocities or densities (pressures) at planar boundaries, the Zou-He method[60] is available in DL_MESO_LBE. This is based upon applying the bounce-back rule to the non-equilibrium distribution functions, i.e.

$$f_i^{(1)}(\vec{x}_w, t) = f_j^{(1)}(\vec{x}_w, t) \tag{5.13}$$

where $f_i^{(1)} = f_i - f_i^{eq}$, with the equilibrium distribution function f_i^{eq} as a function of density and velocity. This function can be used to determine the missing wall velocity or density along with the known distribution function values. For instance, for a top edge with a known velocity $\vec{v}_w$ using the D2Q9 lattice scheme (V??CEDF)[1], the wall density and missing distribution functions (all for the boundary node at $\vec{x}_w$) are given as:

$$\begin{aligned}
\rho_w &= \frac{f_0 + f_2 + f_6 + 2(f_1 + f_7 + f_8)}{1 + v_{w,y}} \\
f_4 &= f_8 - \frac{2\rho_w v_{w,y}}{3} \\
f_3 &= f_7 + \tfrac{1}{2}(f_6 - f_2) - \tfrac{1}{2}\rho_w v_{w,x} - \tfrac{1}{6}\rho_w v_{w,y} \\
f_5 &= f_1 - \tfrac{1}{2}(f_6 - f_2) + \tfrac{1}{2}\rho_w v_{w,x} - \tfrac{1}{6}\rho_w v_{w,y}
\end{aligned}$$

The other form, specifying the wall fluid density, requires the calculation of the wall velocity, which can be simplified by setting non-orthogonal velocity components to zero. For the analogous example at the top wall for D2Q9 (P??CEDF), the same equations for f_4, f_3 and f_5 can be used together with

$$\rho_w v_{w,x} = 0, \rho_w v_{w,y} = f_0 + f_2 + f_6 + 2(f_1 + f_7 + f_8) - \rho_w.$$

[1] '?' denotes any valid letters for the solute composition and temperature.

One complication for three-dimensional lattices is the requirement to apply the non-equilibrium bounce-back to all unknown distribution functions, which ordinarily overspecifies the system but can be counteracted using transverse momentum corrections for directions other than orthogonal to the boundary, which are non-zero for e.g. shearing flows. It should be noted that if the wall velocity is set to zero, the boundary condition reduces to on-grid bounce-back.

DL_MESO_LBE includes implementations of the Zou-He boundary condition for all four lattice schemes: D2Q9, D3Q15, D3Q19 and D3Q27. Only planar surfaces can be dealt with using this method; concave edges (in three-dimensions) and concave corners instead use the equilibrium distribution function for the given density/velocity and either zero velocity or the density at the nearest fluid grid point.

5.2.5 Constant solute concentration/temperature

To specify constant solute concentrations or temperatures at planar boundaries, the Inamuro method[21, 20] is used in DL_MESO_LBE. This is based upon substituting the unknown distribution functions for a boundary point with local equilibrium values, but using an adjusted solute concentration or temperature to produce the correct value for the property at that point. This concentration/temperature is obtained by substituting the equilibrium distribution function for the unknown populations into the sum of distribution functions (equal to the required concentration/temperature).

For example, for a top edge with a specified temperature T_w using the D2Q9 lattice scheme (??TCEDF), the adjustment temperature T' is given as:

$$T' = \frac{6}{1 - 3v_{w,y}} \left(T_w - h_0 - h_1 - h_2 - h_6 - h_7 - h_8\right)$$

and the missing populations ($i = 3, 4, 5$ in this case) are given by

$$h_i = w_i T' \left[1 + 3\frac{(\hat{e}_i \cdot \vec{v}_w)}{c^2}\right]$$

where $\vec{v}_w$ is the known (or calculated) velocity for all fluids at the same boundary.

DL_MESO_LBE includes implementations of the Inamuro boundary condition for all four lattice schemes. Only planar surfaces can be dealt with using this method; as for constant pressure boundaries, concave edges and corners use the equilibrium distribution function for the required concentration/temperature and zero velocity. On-grid bounce-back is used for Neumann (zero-gradient) conditions of solute concentrations and/or temperature to keep this type of boundary condition entirely local.

5.3 Mesoscale interactions

DL_MESO_LBE uses the Shan-Chen model[43, 44] to model interactions between multiple phases and components. This method is based on calculating pairwise interaction potentials by using an 'effective mass' for each component, ψ^a, which is a function of density and most frequently defined as

$$\psi^a\left(\vec{x}\right) = 1 - \exp\left(-\rho^a\left(\vec{x}\right)\right) \tag{5.14}$$

where ρ^a is the density of component a. This can be changed by the user by modifying the subroutine `fCalcPotential_ShanChen`.

Defining g_{ab} as the interaction coefficient between components a and b, the overall force on component a due to interactions with other components is defined as

$$\vec{F}^a = -\psi^a\left(\vec{x}\right) \sum_b g_{ab} \sum_i w_i \psi^b\left(\vec{x} + \hat{e}_i\right) \hat{e}_i. \tag{5.15}$$

and any suitable forcing algorithm can be used to apply this force on a Lattice Boltzmann Equation simulation. For a particular interaction the resulting equation of state[41] is defined as

$$P = \rho c_s^2 + \frac{1}{2} c_s^2 g_{ab} \psi^2(\rho) \tag{5.16}$$

and the interfacial tension between the phases[42] is given as

$$\sigma_{ab} = -\frac{e_4 g_{ab} c_s^4 (\Delta x)^2}{2} \int_{-\infty}^{+\infty} \left(\frac{d\psi}{dz} \right)^2 dz \tag{5.17}$$

where e_4 is a lattice-dependent constant and z distance along the normal from the phase interface.

5.4 Diffusion and heat transfer

In a similar fashion to multiple fluid systems, the Lattice Boltzmann Equation method can be applied to problems involving diffusion and/or heat transfers by using additional distribution functions for each solute and/or temperature[20, 59].

For a system consisting of a number of dilute solutes along with a bulk fluid, the governing equation for each solute is given as

$$g_i \left(\vec{x} + \hat{e}_i \Delta t, t + \Delta t \right) - g_i \left(\vec{x}, t \right) = -\frac{\Delta t}{\tau_s} \left[g_i \left(\vec{x}, t \right) - g_i^{eq} \right] \tag{5.18}$$

where g_i is the distribution function for the solute and τ_s the solute relaxation time, which is related to its diffusivity

$$D = \frac{1}{3} \left(\tau_s - \frac{1}{2} \right) \frac{(\Delta x)^2}{\Delta t}$$

and the Schmidt number can be determined by

$$\mathrm{Sc} = \frac{\nu}{D} = \frac{2\tau_f - 1}{2\tau_s - 1}.$$

Taking the concentration of the solute as $C_s = \sum_i g_i$, the equilibrium distribution function for the solute is given by a simpler form of Equation 3.11:

$$g_i^{eq} = w_i C_s \left[1 + 3 \frac{(\hat{e}_i \cdot \vec{u})}{c^2} \right] \tag{5.19}$$

where the velocity used is that of the bulk fluid.

Heat transfers can be coupled to the system in a similar manner, using a thermal distribution function h_i and a thermal relaxation time τ_t, which gives the governing equation

$$h_i \left(\vec{x} + \hat{e}_i \Delta t, t + \Delta t \right) - h_i \left(\vec{x}, t \right) = -\frac{\Delta t}{\tau_t} \left[h_i \left(\vec{x}, t \right) - h_i^{eq} \right] \tag{5.20}$$

The temperature at each lattice point (relative to a mean value) can be determined as the sum of the distribution functions, $T = \sum_i h_i$, which can be used to determine the equilibrium distribution function

$$h_i^{eq} = w_i T \left[1 + 3 \frac{(\hat{e}_i \cdot \vec{u})}{c^2} \right], \tag{5.21}$$

again using the bulk fluid velocity. The thermal relaxation time is related to the thermal diffusivity

$$\alpha = \frac{1}{3} \left(\tau_t - \frac{1}{2} \right) \frac{(\Delta x)^2}{\Delta t}$$

with the Prandtl number for the system determined by a ratio of relaxation times, i.e.

$$\mathrm{Pr} = \frac{\nu}{\alpha} = \frac{2\tau_f - 1}{2\tau_t - 1}.$$

It should be noted that if multiple-relaxation-time (MRT) schemes are to be used, these only apply to fluids: all diffusion and heat transfer processes are calculated using the single relaxation time schemes described in this section.

5.5 Compressible and incompressible fluids

The standard Lattice Boltzmann Equation scheme is capable of modelling compressible fluids. Incompressible fluids can be modelled by making a simple modification to the local equilibrium distribution function[60]:

$$f_i^{eq} = w_i \left[\rho + \rho_0 \left(\frac{3\left(\hat{e}_i \cdot \vec{u}\right)}{c^2} + \frac{9\left(\hat{e}_i \cdot \vec{u}\right)^2}{2c^4} - \frac{3u^2}{2c^2} \right) \right] \tag{5.22}$$

where ρ_0 is the fixed density of the incompressible fluid and the density ρ becomes an analogue to pressure ($\frac{P}{\rho_0} = c_s^2 \rho$). While Equation 3.1 is still applicable to calculate ρ, the fluid velocity is now calculated by

$$\rho_0 u_\alpha = \sum_{i=0}^{q} f_i e_{i\alpha} \tag{5.23}$$

The initialization, collision and file output routines therefore need to be modified for incompressible fluid systems. DL_MESO_LBE supplies additional routines ending in `Incom` to allow the user to model incompressible fluids, e.g. the subroutine call for the default BGK collision scheme `fCollisionBGK` can be changed to `fCollisionBGKIncom`. All of these routines use a specified constant density for each fluid in the system (`lbincp`) as the value of ρ_0.

Chapter 6

DL_MESO_LBE Package Reference

6.1 Overview

DL_MESO_LBE consists of seven packages:

- `lbpMODEL`
  Contains subroutines to assign `lbw`, `lbv` and `lbopv` for D2Q9, D3Q15, D3Q19 and D3Q27 lattice models.
- `lbpBASIC`
  Contains general purpose subroutines which can be used elsewhere, e.g. a random number generator producing values between -1 and 1.
- `lbpGET`
  Contains subroutines to calculate macroscopic quantities, e.g. macroscopic density, speed and momentum.
- `lbpIO*`
  Contains subroutines to read parameters and write numerical results for plotting and visualization:
  - `lbpIO`
    Contains subroutines to read input files, calculate and write summaries.
  - `lbpIOPlot3D`
    Contains subroutines to write calculation output files in Plot3D format.
  - `lbpIOLegacyVTK`
    Contains subroutines to write calculation output files in Legacy VTK (structured grid) format.
  - `lbpIOVTK`
    Contains subroutines to write calculation output files in XML VTK (structured grid) format.
- `lbpBOUND`
  Contains subroutines for boundary conditions, e.g. calculating the particle distribution function in a shear boundary.
- `lbpSUB`
  Contains the most important subroutines for Lattice Boltzmann calculations, e.g. particle propagation and site collision.
- `lbpMPI`
  Contains all subroutines necessary for parallel computation. This package can be left out if the user uses only a single-processor workstation or a Windows PC.

It is recommended that DL_MESO users put self-defined subroutines into a package called `lbpUSER` so upgrades of DL_MESO will not interfere with their contributions.

6.2 DL_MESO_LBE Subroutines and Functions

6.2.1 main

There are two primary versions of the main DL_MESO_LBE program: serial (`slbe.cpp`) and parallel (`plbe.cpp`). The default listings for these codes are shown below: these model systems using single relaxation time collisions with mesoscopic interactions and stable boundary conditions, and produce outputs in VTK format with densities for fluid 0. An additional version of the serial program is available which uses a boundary layer of lattice points (`slbecombine.cpp`), although the code is not shown here. The user may wish to modify the listings to use different outputs and formats, alternative collision routines, boundary conditions that move through the system etc. and should refer to Appendix A for more details.

Serial code (slbe)

```
#include "slbe.hpp"

int main(int argc, char* argv[])
{
  bigend = fBigEndian();
  fDefineSystem();
  fSetSerialDomain();
  fStartDLMESO();
  fMemoryAllocation();
  fInputParameters();
  fReadSpaceParameter();
  fGetModel();
  fInitializeSystem();
  fReadInitialState();
//  fsOutputGrid();  // Only needed for Plot3D output files
  timetotal=fCheckTimeSerial();
  for(int i=0; i<= lbtotstep; i++) {
    if(i%lbsave  == 0) {
      fsOutputVTK();  // Substitute with required file type and/or output
      qVersion++;
      printf("%d \t ", i);
      fPrintDomainMass();
      fPrintDomainMomentum();
    }
    fInteractionForceShanChen();
    fCollisionBGK();
    fPostCollBoundary();
    fPropagationSwap();
    fPostPropBoundary();
  }
  timetotal=fCheckTimeSerial();
  fFreeMemory();
  fFinishDLMESO();
  return 0;
}
```

Parallel code (plbe)

```
#include "plbe.hpp"

int main(int argc, char** argv)
{
  bigend = fBigEndian();
  fStartMPI(argc, argv);
  fDefineSystem("lbin.sys");
  fDefineDomain();
  fStartDLMESO();
  fMemoryAllocation();
  fDefineNeighbours();
  fDefineMessage();
  fPrintSystemInfo();
  fAllReady();
  fPrintDomainInfo();
  fAllReady();
  fInputParameters("lbin.sys");
  fReadSpaceParameter("lbin.spa");
  fGetModel();
  fBoundNonBlockCommunication();
  fMarkBoundArea();
  fInitializeSystem();
  fReadInitialState("lbin.init");
  fAllReady();
  fOutputInfo();
//  fOutputGrid();  // Only needed for Plot3D output files
  timetotal=fCheckTimeMPI();
  for(int i=0; i<= lbtotstep; i++) {
    fNonBlockCommunication();
    if(i%lbsave  == 0) {
      fOutputVTK();  // Substitute with required file type and/or output
      qVersion++;
      if(lbdm.rank == 0)
printf("%d \t ", i);
      fPrintSystemMass();
      fPrintSystemMomentum();
    }
    fInteractionForceShanChen();
    fForceNonBlockCommunication();
    fCollisionBGK();
    fPostCollBoundary();
    fPropagationCombSwap();
    fPostPropBoundary();
  }
  timetotal=fCheckTimeMPI();
  fFreeMemory();
  fFinishDLMESO();
  fCloseMPI();
  return 0;
```

}

6.2.2 lbpMODEL

D2Q9

- **Header records**
  `int D2Q9()`
- **Function**
  Assign the weight factor `lbw`, speed vector `lbv`, index for opposite speed vector `lbopv`, MRT transformation matrices (`lbtr` and `lbtrinv`) and tuneable parameters (`lbmrts` and `lbmrtw`) for the D2Q9 lattice model.
- **Dependencies**
  None

D3Q15

- **Header records**
  `int D3Q15()`
- **Function**
  Assign the weight factor `lbw`, speed vector `lbv`, index for opposite speed vector `lbopv`, MRT transformation matrices (`lbtr` and `lbtrinv`) and tuneable parameters (`lbmrts` and `lbmrtw`) for the D3Q15 lattice model.
- **Dependencies**
  None

D3Q19

- **Header records**
  `int D3Q19()`
- **Function**
  Assign the weight factor `lbw`, speed vector `lbv`, index for opposite speed vector `lbopv`, MRT transformation matrices (`lbtr` and `lbtrinv`) and tuneable parameters (`lbmrts` and `lbmrtw`) for the D3Q19 lattice model.
- **Dependencies**
  None

D3Q27

- **Header records**
  `int D3Q27()`
- **Function**
  Assign the weight factor `lbw`, speed vector `lbv` and the index for opposite speed vector `lbopv` for the D3Q27 lattice model.
- **Dependencies**
  None
- **Comments**
  No Multiple-Relaxation-Time (MRT) scheme is currently available for this lattice.

6.2.3 lbpBASIC

This package contains general purpose functions which are not directly related to the Lattice Boltzmann Equation method. These may be replaced with any suitable functions in C++ standard libraries.

fCppAbs

- **Header records**
  `template <class T>`
  `T fCppAbs(T a)return (a<0)?-a:a;`
- **Function**
  Calculate absolute value of numerical number `a`.
- **Dependencies**
  None
- **Arguments**

`a`	input	any datatype
`fCppAbs`	output	same datatype as `a`

fSwapPair

- **Header records**
  `template <class T> void fSwapPair ( T& a, T& b)`
- **Function**
  Swaps pair of numbers, `a` and `b`.
- **Dependencies**
  None
- **Arguments**

`a`	input/output	any datatype
`b`	input/output	any datatype

fGetNumberOrdered

- **Header records** (two cases)
  `int fGetNumberOrdered(int &iox, int &ioy, int &ioz)`
  `int fGetNumberOrdered(int &iox, int &ioy)`
- **Function**
  Rearrange the integers in descending order.
- **Dependencies**
  None
- **Arguments**

`iox`	input/output	integer reference
`ioy`	input/output	integer reference
`ioz`	input/output	integer reference

fGetNumberOrderFixed

- **Header records** (two cases)
  `int fGetNumberOrderFixed(int &iox, int &ioy, int &ioz, int ix, int iy, int iz)`
  `int fGetNumberOrderFixed(int &iox, int &ioy, int ix, int iy)`

- **Function**
  Rearrange a set of integers so they appear in the same order as another set of integers.

- **Dependencies**
  `fGetNumberOrdered`

- **Arguments**

`iox`	input/output	integer reference
`ioy`	input/output	integer reference
`ioz`	input/output	integer reference
`ix`	input	integer
`iy`	input	integer
`iz`	input	integer

fBestGrouping

- **Header records**
  `int fBestGrouping(int totalgrid, int totalgroup, int& indigrid, int& critigroup)`

- **Function**
  Distribute grid points among the processes to give a maximum of one to the differences in the numbers of grid points.

- **Dependencies**
  None

- **Arguments**

`totalgrid`	input	integer
`totalgroup`	input	integer
`indigrid`	output	integer reference
`critigroup`	output	integer reference

- **Comments**
  The `totalgrid` grid points are distributed among `totalgroup` processes so that the first `critigroup` processes have `indigrid` grid points and the others have `indigrid-1`.

fCppMod

- **Header records** (two cases)
  `int fCppMod(int a, int b)`
  `long fCppMod(long a, long b)`

- **Function**
  Ensure that `a` is in a range between `0` and `b-1`, so that the value beyond the maximum value equals the minimum, and vice versa.

- **Dependencies**
  None

- **Arguments**

`a`	input	integer/long integer
`b`	input	integer/long integer
`fCppMod`	output	integer/long integer

- **Comments**
  `fCppMod = 0` when `a = b` or `fCppMod = b-1` when `a = -1`. This function is useful for periodic boundary conditions.

fPrintLine

- **Header records**
  `int fPrintLine()`
- **Function**
  Prints a line of 72 `-` characters.
- **Dependencies**
  None

fPrintDoubleLine

- **Header records**
  `int fPrintDoubleLine()`
- **Function**
  Prints a line of 72 `=` characters.
- **Dependencies**
  None

fRandom

- **Header records**
  `double fRandom()`
- **Function**
  Creates a double precision random number between -1 and 1.
- **Dependencies**
  None
- **Arguments**

`fRandom`	output	double precision

- **Comments**
  There is a built-in seed in the function, which is only activated when the function is initially called.

fBigEndian

- **Header records**
  `int fBigEndian()`
- **Function**
  Determines endianness for machine: returns 1 for big endian, 0 for little endian.

- **Dependencies**
  None

fByteSwap

- **Header records**
  `void fByteSwap(void *data, int len, int count)`
- **Function**
  Converts between endian types by swapping byte order of array `data` (with byte size per entry `len` and `count` entries).
- **Dependencies**
  None
- **Arguments**

`data`	input/output	void
`len`	input	integer
`count`	input	integer

- **Comments**
  Primarily required for writing binary files where a specific endianness is required, e.g. `.vtk` files are required in big endian.

fCheckTimeSerial

- **Header records**
  `double fCheckTimeSerial()`
- **Function**
  Outputs time in seconds since initial call.
- **Dependencies**
  None
- **Arguments**

`fCheckTimeSerial`	output	double

- **Comments**
  Obtains calculation time based on system clock; parallel calculations may obtain greater timing accuracy with `fCheckTimeMPI`.

6.2.4 lbpGET

fGetNodePosi

- **Header records** (two cases: 3D and 2D)
  `inline long fGetNodePosi(int xpos, int ypos, int zpos)`
  `inline long fGetNodePosi(int xpos, int ypos)`
- **Function**
  Calculates the position of the grid point in a one-dimensional array from its Cartesian coordinate.
- **Dependencies**
  None

- **Arguments**

`xpos`	input	integer
`ypos`	input	integer
`zpos`	input	integer
`fGetNodePosi`	output	long integer

- **Comments**
  The calculation follows the standard C++ data structure (row-major).

fGetCoord

- **Header records** (two cases: 3D and 2D)
  `int fGetCoord(long tpos, int& xpos, int& ypos, int& zpos)`
  `int fGetCoord(long tpos, int& xpos, int& ypos)`

- **Function**
  Calculates the Cartesian coordinate of a grid point from its position in a one-dimensional array.

- **Dependencies**
  None

- **Arguments**

`xpos`	output	integer reference
`ypos`	output	integer reference
`zpos`	output	integer reference
`tpos`	input	long integer

fGetOneMassSite

- **Header records** (three cases)
  `double fGetOneMassSite(double* startpos)`
  `double fGetOneMassSite(int fpos, long tpos)`
  `double fGetOneMassSite(int fpos, int xpos, int ypos, int zpos)`

- **Function**
  Calculates the mass density of one of the fluids at a grid point.

- **Dependencies**
  None

- **Arguments**

`startpos`	input	double pointer
`fpos`	input	integer
`tpos`	input	long integer
`xpos`	input	integer
`ypos`	input	integer
`zpos`	input	integer
`fGetOneMassSite`	output	double precision

- **Comments**
  Mass density is calculated according to the definition $\rho = \sum_i f_i$. In the first case, `startpos` is the start point for the summation of particle distribution functions and must be assigned correctly. The second case carries out the same calculation for the `fpos`-th fluid and `tpos`-th grid point, while the third carries it out for the `fpos`-th fluid at the grid point indicated by the Cartesian coordinate (`xpos`, `ypos`, `zpos`). The latter two are more readable to the user but a bit slower to carry out.

fGetTotMassSite

- **Header records** (two cases)
  `double fGetTotMassSite(double* startpos)`
  `double fGetTotMassSite(long tpos)`
- **Function**
  Calculates the mass density of all fluids at a grid point.
- **Dependencies**
  None
- **Arguments**

`startpos`	input	double pointer
`tpos`	input	long integer
`fGetTotMassSite`	output	double precision

- **Comments**
  Mass density is calculated according to the definition $\rho = \sum_i f_i$. The second case carries out the same calculation as the first but using `tpos` for the grid point.

fGetOneMassDomain

- **Header records**
  `double fGetOneMassDomain(int fpos)`
- **Function**
  Calculates the total mass of `fpos`-th fluid in the domain.
- **Dependencies**
  None
- **Arguments**

`fpos`	input	integer
`fGetOneMassDomain`	output	double precision

- **Comments**
  The total mass of the domain does not include boundary areas used for message passing or nodes used to apply boundary conditions.

fGetTotMassDomain

- **Header records**
  `double fGetTotMassDomain()`
- **Function**
  Calculates the total mass of all fluids in the domain.
- **Dependencies**
  None
- **Arguments**

`fGetTotMassDomain`	output	double precision

- **Comments**
  The total mass of the domain does not include boundary areas used for message passing or nodes used to apply boundary conditions.

fGetFracSite

- **Header records** (three cases)
  `double fGetFracSite(int fpos, double* startpos)`
  `double fGetFracSite(int fpos, long tpos)`
  `double fGetFracSite(int fpos, int xpos, int ypos, int zpos)`
- **Function**
  Calculates the mass fraction of fluid `fpos` in the site.
- **Dependencies**
  None
- **Arguments**

`startpos`	input	double pointer
`fpos`	input	integer
`tpos`	input	long integer
`xpos`	input	integer
`ypos`	input	integer
`zpos`	input	integer
`fGetFracSite`	output	double precision

- **Comments**
  The calculation is based on $z = \frac{\rho_i}{\sum_i \rho_i}$. This function operates in a similar way to `fGetOneMassSite` with the second and third cases slightly slower than the first but more readable.

fGetOneConcSite

- **Header records** (two cases)
  `double fGetOneConcSite(int cpos, long tpos)`
  `double fGetOneConcSite(int cpos, int xpos, int ypos, int zpos)`
- **Function**
  Calculates the concentration of solute `cpos` at the grid point.
- **Dependencies**
  `fGetOneMassSite`
- **Arguments**

`tpos`	input	long integer
`cpos`	input	integer
`xpos`	input	integer
`ypos`	input	integer
`zpos`	input	integer
`fGetFracSite`	output	double precision

fGetTemperatureSite

- **Header records** (two cases)
  `double fGetTemperatureSite(long tpos)`
  `double fGetTemperatureSite(long xpos, long ypos, long zpos)`
- **Function**
  Calculates the scalar temperature at the grid point.

- **Dependencies**
  `fGetOneMassSite`

- **Arguments**

`tpos`	input	long integer
`xpos`	input	long integer
`ypos`	input	long integer
`zpos`	input	long integer
`fGetFracSite`	output	double precision

fGetOneSpeedSite

- **Header records** (three cases)
```
int fGetOneSpeedSite(double *speed, double* startpos)
int fGetOneSpeedSite(double *speed, int fpos, long tpos)
int fGetOneSpeedSite(double *speed, int fpos, int xpos, int ypos, int zpos)
```

- **Function**
  Calculates the macroscopic speed of (compressible) fluid `fpos` at the grid point.

- **Dependencies**
  None

- **Arguments**

`startpos`	input	double pointer
`fpos`	input	integer
`tpos`	input	long integer
`xpos`	input	integer
`ypos`	input	integer
`zpos`	input	integer
`speed`	output	double precision array

- **Comments**
  The calculation is based on $v_\alpha = \frac{\sum_i f_i e_{i\alpha}}{\rho}$. The second and third cases are slightly slower than the first but more readable.

fGetOneSpeedIncomSite

- **Header records** (three cases)
```
int fGetOneSpeedIncomSite(double *speed, double* startpos, double rho0)
int fGetOneSpeedIncomSite(double *speed, int fpos, long tpos)
int fGetOneSpeedIncomSite(double *speed, int fpos, int xpos, int ypos, int zpos)
```

- **Function**
  Calculates the macroscopic speed of incompressible fluid `fpos` at the grid point.

- **Dependencies**
  None

- **Arguments**

`startpos`	input	double pointer
`rho0`	input	double precision
`fpos`	input	integer
`tpos`	input	long integer
`xpos`	input	integer
`ypos`	input	integer
`zpos`	input	integer
`speed`	output	double precision array

- **Comments**
  The calculation is based on $v_\alpha = \frac{\sum_i f_i e_{i\alpha}}{\rho_0}$. The second and third cases are slightly slower than the first but more readable.

fGetSpeedSite

- **Header records** (three cases)

```
int fGetSpeedSite(double *speed, double* startpos)
int fGetSpeedSite(double *speed, long tpos)
int fGetSpeedSite(double *speed, int xpos, int ypos, int zpos)
```

- **Function**
  Calculates the macroscopic speed of all (compressible) fluids at the grid point.

- **Dependencies**
  None

- **Arguments**

`startpos`	input	double pointer
`tpos`	input	long integer
`xpos`	input	integer
`ypos`	input	integer
`zpos`	input	integer
`speed`	output	double precision array

- **Comments**
  The calculation is based on $v_\alpha = \frac{\sum_i f_i e_{i\alpha}}{\rho}$. The second and third cases are more readable than the first but also slightly slower.

fGetSpeedIncomSite

- **Header records** (three cases)

```
int fGetSpeedIncomSite(double *speed, double* startpos)
int fGetSpeedIncomSite(double *speed, long tpos)
int fGetSpeedIncomSite(double *speed, int xpos, int ypos, int zpos)
```

- **Function**
  Calculates the macroscopic speed of all incompressible fluids at the grid point.

- **Dependencies**
  None

- **Arguments**

`startpos`	input	double pointer
`tpos`	input	long integer
`xpos`	input	integer
`ypos`	input	integer
`zpos`	input	integer
`speed`	output	double precision array

- **Comments**
  The calculation is based on $v_\alpha = \frac{\sum_i f_i e_{i\alpha}}{\rho_0}$. The second and third cases are more readable than the first but also slightly slower.

fGetOneMomentSite

- **Header records** (three cases)
  `int fGetOneMomentSite(double *speed, double* startpos)`
  `int fGetOneMomentSite(double *speed, int fpos, long tpos)`
  `int fGetOneMomentSite(double *speed, int fpos, int xpos, int ypos, int zpos)`

- **Function**
  Calculates the momentum of one of the fluids at the grid point.

- **Dependencies**
  None

- **Arguments**

`startpos`	input	double pointer
`fpos`	input	integer
`tpos`	input	long integer
`xpos`	input	integer
`ypos`	input	integer
`zpos`	input	integer
`speed`	output	double precision array

- **Comments**
  The calculation is based on $p_\alpha = \sum_i f_i e_{i\alpha}$. The second and third cases are more readable than the first but also slightly slower.

fGetTotMomentSite

- **Header records**
  `int fGetTotMomentSite(double *speed, double* startpos)`

- **Function**
  Calculates the momentum of all fluids at the grid point.

- **Dependencies**
  None

- **Arguments**

`startpos`	input	double pointer
`speed`	output	double precision array

fGetTotMomentDomain

- **Header records**
  `int fGetTotMomentDomain(double *momentum)`
- **Function**
  Calculates the momentum of all fluids in the domain.
- **Dependencies**
  `fGetTotMomentSite`
- **Arguments**
  `momentum` output double precision array
- **Comments**
  This function is mainly used to verify that the domain momentum along each axis is conserved.

fGetOneDirecSpeedSite

- **Header records** (three cases)
  `float fGetOneDirecSpeedSite(int dire, double* startpos)`
  `float fGetOneDirecSpeedSite(int dire, long tpos)`
  `float fGetOneDirecSpeedSite(int dire, int xpos, int ypos, int zpos)`
- **Function**
  Calculates the grid speed for all (compressible) fluids along direction `dire`: `0` for x, `1` for y and `2` for z.
- **Dependencies**
  None
- **Arguments**

`startpos`	input	double pointer
`tpos`	input	long integer
`xpos`	input	integer
`ypos`	input	integer
`zpos`	input	integer
`dire`	input	double pointer
`fGetOneDirecSpeedSite`	output	floating point

- **Comments**
  Mainly used to output grid speed. The second and third cases are more readable than the first but also slightly slower.

fGetOneDirecSpeedIncomSite

- **Header records** (three cases)
  `float fGetOneDirecSpeedIncomSite(int dire, double* startpos)`
  `float fGetOneDirecSpeedIncomSite(int dire, long tpos)`
  `float fGetOneDirecSpeedIncomSite(int dire, int xpos, int ypos, int zpos)`
- **Function**
  Calculates the grid speed for all incompressible fluids along direction `dire`: `0` for x, `1` for y and `2` for z.
- **Dependencies**
  None

- **Arguments**

`startpos`	input	double pointer
`tpos`	input	long integer
`xpos`	input	integer
`ypos`	input	integer
`zpos`	input	integer
`dire`	input	double pointer
`fGetOneDirecSpeedIncomSite`	output	floating point

- **Comments**
  Mainly used to output grid speed. The second and third cases are more readable than the first but also slightly slower.

6.2.5 lbpIO

fDefineSystem

- **Header records**
  `int fDefineSystem(const char* filename = "lbin.sys")`
- **Function**
  Reads calculation parameters (lattice scheme, numbers of fluids, solutes, temperature scalars and phase field order parameters, grid size) from input system file `lbin.sys`.
- **Dependencies**
  `lbin.sys` data file
- **Arguments**

`filename`	input	array of characters

- **Comments**
  The default file name is `lbin.sys`: to use different file names the argument for this routine can be changed by the user.

fPrintDomainMass

- **Header records**
  `int fPrintDomainMass()`
- **Function**
  Calculates and prints total and individual fluid masses in domain.
- **Dependencies**
  `double fGetTotMassDomain()`
  `double fGetOneMassDomain()`
- **Comments**
  This routine only produces the masses for the entire system if running in serial; to obtain system masses in parallel running, `fPrintSystemMass()` would be needed.

fPrintDomainMomentum

- **Header records**
  `int fPrintDomainMomentum()`

- **Function**
  Calculates and prints the total fluid momentum in domain.
- **Dependencies**
  `int fGetTotMomentDomain(double *momentum)`
- **Comments**
  This routine only produces the entire system momentum if running in serial; to obtain system momentum in parallel running, `fPrintSystemMomentum()` would be needed.

fInputParameters

- **Header records**
  `int fInputParameters(const char* filename="lbin.sys")`
- **Function**
  Reads system parameters in from `lbin.sys` data file.
- **Dependencies**
  `lbin.sys` data file
- **Arguments**
  `filename` input array of characters
- **Comments**
  The default file name is `lbin.sys`: to use different file names the argument for this routine can be changed by the user.

fReadSpaceParameter

- **Header records**
  `int fReadSpaceParameter(const char* filename="lbin.spa")`
- **Function**
  Reads 2D or 3D space parameters from data file `lbin.spa`.
- **Dependencies**
  `lbin.spa` data file
  `fReadSpace2D`
  `fReadSpace3D`
- **Arguments**
  `filename` input array of characters
- **Comments**
  The default file name is `lbin.spa`: this can be changed by the user if the space data file has a different name.

fReadInitialState

- **Header records**
  `int fReadInitialState(const char* filename="lbin.init")`
- **Function**
  Reads initial velocities, densities, concentrations and temperatures from data file `lbin.init` and calculates initial distribution functions for compressible fluids.

- **Dependencies**
  `lbin.init` data file
  `fReadInitialState2D`
  `fReadInitialState3D`

- **Arguments**
  `filename` input array of characters

- **Comments**
  The default file name is `lbin.init`: this can be changed by the user if the initial state data file has a different name. This routine will replace the default values of all properties at specified points and should be called after `fInitializeSystem`.

fReadInitialStateIncom

- **Header records**
  `int fReadInitialStateIncom(const char* filename="lbin.init")`

- **Function**
  Reads initial velocities, densities, concentrations and temperatures from data file `lbin.init` and calculates initial distribution functions for incompressible fluids.

- **Dependencies**
  `lbin.init` data file
  `fReadInitialState2DIncom`
  `fReadInitialState3DIncom`

- **Arguments**
  `filename` input array of characters

- **Comments**
  The default file name is `lbin.init`: this can be changed by the user if the initial state data file has a different name. This routine will replace the default values of all properties at specified points and should be called after `fInitializeSystemIncom`.

fSetoffSteer

- **Header records**
  `int fSetoffSteer()`

- **Function**
  Creates a file called `notsteer` to prevent DL_MESO_LBE from creating new `lbin.sys` and `lbin.spa` files (which occurs if `notsteer` is missing).

- **Dependencies**
  None

- **Comments**
  If the user has changed the input datafiles, the code will run with new parameters.

fCheckSteer

- **Header records**
  `int fCheckSteer()`

- **Function**
  Checks for the existence of `notsteer` files: if found, then reads `lbin.sys` and `lbin.spa` files.

- **Dependencies**
  `fInputParameters("lbin.sys")`
  `fReadSpaceParameter("lbin.spa")`

6.2.6 lbpIOPlot3D

fOutputGrid

- **Header records**
  `int fOutputGrid3D(const char* filename="lbout")`

- **Function**
  Outputs grid positions for system in Plot3D format.

- **Dependencies**
  `fOutputGrid2D`
  `fOutputGrid3D`

- **Arguments**
  `filename` input array of characters

- **Comments**
  The default output file name is `lbout*.xyz` for 3D systems and `lbout*.xy` for 2D systems.

fOutputQ

- **Header records**
  `int fOutputQ(const char* filename="lbout", int iprop=0)`

- **Function**
  Outputs macroscopic mass density and velocity (speeds along x-, y- and z-directions) at each lattice point of compressible fluid `iprop` for system in Plot3D format.

- **Dependencies**
  `fOutputQ2D`
  `fOutputQ3D`

- **Arguments**
  `filename` input array of characters

- **Comments**
  The default output file name is `lbout*.q` and the density of fluid 0 is output by default. This can be changed by specifying an output file name and fluid number (up to `lbsy.nf`-1) when calling the routine.

fOutputQIncom

- **Header records**
  `int fOutputQIncom(const char* filename="lbout", int iprop=0)`

- **Function**
  Outputs macroscopic mass density and velocity (speeds along x-, y- and z-directions) at each lattice point of incompressible fluid `iprop` for system in Plot3D format.

- **Dependencies**
  `fOutputQ2DIncom`
  `fOutputQ3DIncom`

- **Arguments**
  `filename` input array of characters

- **Comments**
  The default output file name is `lbout*.q` and the density of fluid 0 is output by default. This can be changed by specifying an output file name and fluid number (up to `lbsy.nf`−1) when calling the routine.

fOutputQCA

- **Header records**
  `int fOutputQCA(const char* filename="lbout", int iprop=0)`

- **Function**
  Outputs mass fraction and speeds along x-, y- and z-directions of compressible fluid `iprop` at each lattice point for system in Plot3D format.

- **Dependencies**
  `fOutputQCA2D`
  `fOutputQCA3D`

- **Arguments**
  `filename` input array of characters

- **Comments**
  The default output file name is `lbout*.q` and the mass fraction of fluid 0 is output by default. These can be changed by specifying an output file name and fluid number (up to `lbsy.nf`−1) when calling the routine.

fOutputQCAIncom

- **Header records**
  `int fOutputQCAIncom(const char* filename="lbout", int iprop=0)`

- **Function**
  Outputs mass fraction and speeds along x-, y- and z-directions of incompressible fluid `iprop` at each lattice point for system in Plot3D format.

- **Dependencies**
  `fOutputQCA2DIncom`
  `fOutputQCA3DIncom`

- **Arguments**
  `filename` input array of characters

- **Comments**
  The default output file name is `lbout*.q` and the mass fraction of fluid 0 is output by default. These can be changed by specifying an output file name and fluid number (up to `lbsy.nf`−1) when calling the routine.

fOutputQCB

- **Header records**
  `int fOutputQCB(const char* filename="lbout", int iprop=0)`
- **Function**
  Outputs concentration of solute `iprop` and (compressible fluid) velocity at each lattice point for system in Plot3D format.
- **Dependencies**
  `fOutputQCB2D`
  `fOutputQCB3D`
- **Arguments**
  `filename` input array of characters
- **Comments**
  The default output file name is `lbout*.q` and the concentration of solute 0 is output by default. These can be changed by specifying an output file name and solute number (up to `lbsy.ns`−1) when calling the routine.

fOutputQCBIncom

- **Header records**
  `int fOutputQCBIncom(const char* filename="lbout", int iprop=0)`
- **Function**
  Outputs concentration of solute `iprop` and (incompressible fluid) velocity at each lattice point for system in Plot3D format.
- **Dependencies**
  `fOutputQCB2DIncom`
  `fOutputQCB3DIncom`
- **Arguments**
  `filename` input array of characters
- **Comments**
  The default output file name is `lbout*.q` and the concentration of solute 0 is output by default. These can be changed by specifying an output file name and solute number (up to `lbsy.ns`−1) when calling the routine.

fOutputQT

- **Header records**
  `int fOutputQT(const char* filename="lbout")`
- **Function**
  Outputs macroscopic temperature and (compressible fluid) velocity at each lattice point for system in Plot3D format.
- **Dependencies**
  `fOutputQT2D`
  `fOutputQT3D`

- **Arguments**
  `filename` input array of characters

- **Comments**
  The default output file name is `lbout*.q`. This can be changed by specifying an output file name when calling the routine.

fOutputQTIncom

- **Header records**
  `int fOutputQTIncom(const char* filename="lbout")`

- **Function**
  Outputs macroscopic temperature and (incompressible fluid) velocity at each lattice point for system in Plot3D format.

- **Dependencies**
  `fOutputQT2DIncom`
  `fOutputQT3DIncom`

- **Arguments**
  `filename` input array of characters

- **Comments**
  The default output file name is `lbout*.q`. This can be changed by specifying an output file name when calling the routine.

6.2.7 lbpIOLegacyVTK

fOutputLegacyVTK

- **Header records**
  `int fOutputLegacyVTK(const char* filename="lbout", int iprop=0)`

- **Function**
  Outputs macroscopic mass density and velocity (speeds along x-, y- and z-directions) at each lattice point of compressible fluid `iprop` for system in Structured Grid Legacy VTK format.

- **Dependencies**
  `fOutputLegacyVTK2D`
  `fOutputLegacyVTK3D`

- **Arguments**
  `filename` input array of characters

- **Comments**
  The default output file name is `lbout*.q` and the density of fluid 0 is output by default. This can be changed by specifying an output file name and fluid number (up to `lbsy.nf`-1) when calling the routine.

fOutputLegacyVTKIncom

- **Header records**
  `int fOutputLegacyVTKIncom(const char* filename="lbout", int iprop=0)`

- **Function**
  Outputs macroscopic mass density and velocity (speeds along x-, y- and z-directions) at each lattice point of incompressible fluid `iprop` for system in Structured Grid Legacy VTK format.

- **Dependencies**
  `fOutputLegacyVTK2DIncom`
  `fOutputLegacyVTK3DIncom`

- **Arguments**
  `filename` input array of characters

- **Comments**
  The default output file name is `lbout*.vtk` and the density of fluid 0 is output by default. This can be changed by specifying an output file name and fluid number (up to `lbsy.nf`-1) when calling the routine.

fOutputLegacyVTKCA

- **Header records**
  `int fOutputLegacyVTKCA(const char* filename="lbout", int iprop=0)`

- **Function**
  Outputs mass fraction and speeds along x-, y- and z-directions of compressible fluid `iprop` at each lattice point for system in Structured Grid Legacy VTK format.

- **Dependencies**
  `fOutputLegacyVTKCA2D`
  `fOutputLegacyVTKCA3D`

- **Arguments**
  `filename` input array of characters

- **Comments**
  The default output file name is `lbout*.vtk` and the mass fraction of fluid 0 is output by default. These can be changed by specifying an output file name and fluid number (up to `lbsy.nf`-1) when calling the routine.

fOutputLegacyVTKCAIncom

- **Header records**
  `int fOutputLegacyVTKCAIncom(const char* filename="lbout", int iprop=0)`

- **Function**
  Outputs mass fraction and speeds along x-, y- and z-directions of incompressible fluid `iprop` at each lattice point for system in Structured Grid Legacy VTK format.

- **Dependencies**
  `fOutputLegacyVTKCA2DIncom`
  `fOutputLegacyVTKCA3DIncom`

- **Arguments**
  `filename` input array of characters

- **Comments**
  The default output file name is `lbout*.vtk` and the mass fraction of fluid 0 is output by default. These can be changed by specifying an output file name and fluid number (up to `lbsy.nf`-1) when calling the routine.

fOutputLegacyVTKCB

- **Header records**
  `int fOutputLegacyVTKCB(const char* filename="lbout", int iprop=0)`
- **Function**
  Outputs concentration of solute `iprop` and (compressible fluid) velocity at each lattice point for system in Structured Grid Legacy VTK format.
- **Dependencies**
  `fOutputLegacyVTKCB2D`
  `fOutputLegacyVTKCB3D`
- **Arguments**
  `filename` input array of characters
- **Comments**
  The default output file name is `lbout*.vtk` and the concentration of solute 0 is output by default. These can be changed by specifying an output file name and solute number (up to `lbsy.ns`−1) when calling the routine.

fOutputLegacyVTKCBIncom

- **Header records**
  `int fOutputLegacyVTKCBIncom(const char* filename="lbout", int iprop=0)`
- **Function**
  Outputs concentration of solute `iprop` and (incompressible fluid) velocity at each lattice point for system in Structured Grid Legacy VTK format.
- **Dependencies**
  `fOutputLegacyVTKCB2DIncom`
  `fOutputLegacyVTKCB3DIncom`
- **Arguments**
  `filename` input array of characters
- **Comments**
  The default output file name is `lbout*.q` and the concentration of solute 0 is output by default. These can be changed by specifying an output file name and solute number (up to `lbsy.ns`−1) when calling the routine.

fOutputLegacyVTKT

- **Header records**
  `int fOutputLegacyVTKT(const char* filename="lbout")`
- **Function**
  Outputs macroscopic temperature and (compressible fluid) velocity at each lattice point for system in Structured Grid Legacy VTK format.
- **Dependencies**
  `fOutputLegacyVTKT2D`
  `fOutputLegacyVTKT3D`

- **Arguments**
  `filename` input array of characters
- **Comments**
  The default output file name is `lbout*.vtk`. This can be changed by specifying an output file name when calling the routine.

fOutputLegacyVTKTIncom

- **Header records**
  `int fOutputLegacyVTKTIncom(const char* filename="lbout")`
- **Function**
  Outputs macroscopic temperature and (incompressible fluid) velocity at each lattice point for system in Structured Grid Legacy VTK format.
- **Dependencies**
  `fOutputLegacyVTKT2DIncom`
  `fOutputLegacyVTKT3DIncom`
- **Arguments**
  `filename` input array of characters
- **Comments**
  The default output file name is `lbout*.vtk`. This can be changed by specifying an output file name when calling the routine.

6.2.8 lbpIOVTK

fOutputVTK

- **Header records**
  `int fOutputVTK(const char* filename="lbout", int iprop=0)`
- **Function**
  Outputs macroscopic mass density and velocity (speeds along x-, y- and z-directions) at each lattice point of compressible fluid `iprop` for system in Structured Grid XML VTK format.
- **Dependencies**
  `fOutputVTK2D`
  `fOutputVTK3D`
- **Arguments**
  `filename` input array of characters
- **Comments**
  The default output file name is `lbout*.vts` and the density of fluid 0 is output by default. This can be changed by specifying an output file name and fluid number (up to `lbsy.nf`-1) when calling the routine.

fOutputVTKIncom

- **Header records**
  `int fOutputVTKIncom(const char* filename="lbout", int iprop=0)`

- **Function**
  Outputs macroscopic mass density and velocity (speeds along x-, y- and z-directions) at each lattice point of incompressible fluid `iprop` for system in Structured Grid XML VTK format.

- **Dependencies**
  `fOutputVTK2DIncom`
  `fOutputVTK3DIncom`

- **Arguments**
  `filename` input array of characters

- **Comments**
  The default output file name is `lbout*.vts` and the density of fluid 0 is output by default. This can be changed by specifying an output file name and fluid number (up to `lbsy.nf`-1) when calling the routine.

fOutputVTKCA

- **Header records**
  `int fOutputVTKCA(const char* filename="lbout", int iprop=0)`

- **Function**
  Outputs mass fraction and speeds along x-, y- and z-directions of compressible fluid `iprop` at each lattice point for system in Structured Grid XML VTK format.

- **Dependencies**
  `fOutputVTKCA2D`
  `fOutputVTKCA3D`

- **Arguments**
  `filename` input array of characters

- **Comments**
  The default output file name is `lbout*.vts` and the mass fraction of fluid 0 is output by default. These can be changed by specifying an output file name and fluid number (up to `lbsy.nf`-1) when calling the routine.

fOutputVTKCAIncom

- **Header records**
  `int fOutputVTKCAIncom(const char* filename="lbout", int iprop=0)`

- **Function**
  Outputs mass fraction and speeds along x-, y- and z-directions of incompressible fluid `iprop` at each lattice point for system in Structured Grid XML VTK format.

- **Dependencies**
  `fOutputVTKCA2DIncom`
  `fOutputVTKCA3DIncom`

- **Arguments**
  `filename` input array of characters

- **Comments**
  The default output file name is `lbout*.vts` and the mass fraction of fluid 0 is output by default. These can be changed by specifying an output file name and fluid number (up to `lbsy.nf`-1) when calling the routine.

fOutputVTKCB

- **Header records**
  `int fOutputVTKCB(const char* filename="lbout", int iprop=0)`
- **Function**
  Outputs concentration of solute `iprop` and (compressible fluid) velocity at each lattice point for system in Structured Grid XML VTK format.
- **Dependencies**
  `fOutputVTKCB2D`
  `fOutputVTKCB3D`
- **Arguments**
  `filename` input array of characters
- **Comments**
  The default output file name is `lbout*.vts` and the concentration of solute 0 is output by default. These can be changed by specifying an output file name and solute number (up to `lbsy.ns`−1) when calling the routine.

fOutputVTKCBIncom

- **Header records**
  `int fOutputVTKCBIncom(const char* filename="lbout", int iprop=0)`
- **Function**
  Outputs concentration of solute `iprop` and (incompressible fluid) velocity at each lattice point for system in Structured Grid XML VTK format.
- **Dependencies**
  `fOutputVTKCB2DIncom`
  `fOutputVTKCB3DIncom`
- **Arguments**
  `filename` input array of characters
- **Comments**
  The default output file name is `lbout*.vts` and the concentration of solute 0 is output by default. These can be changed by specifying an output file name and solute number (up to `lbsy.ns`−1) when calling the routine.

fOutputVTKT

- **Header records**
  `int fOutputVTKT(const char* filename="lbout")`
- **Function**
  Outputs macroscopic temperature and (compressible fluid) velocity at each lattice point for system in Structured Grid XML VTK format.
- **Dependencies**
  `fOutputVTKT2D`
  `fOutputVTKT3D`

- **Arguments**
  `filename` input array of characters
- **Comments**
  The default output file name is `lbout*.vts`. This can be changed by specifying an output file name when calling the routine.

fOutputVTKTIncom

- **Header records**
  `int fOutputVTKTIncom(const char* filename="lbout")`
- **Function**
  Outputs macroscopic temperature and (incompressible fluid) velocity at each lattice point for system in Structured Grid XML VTK format.
- **Dependencies**
  `fOutputVTKT2DIncom`
  `fOutputVTKT3DIncom`
- **Arguments**
  `filename` input array of characters
- **Comments**
  The default output file name is `lbout*.vts`. This can be changed by specifying an output file name when calling the routine.

Other output routines

Subroutines with names `fsOutput*` are suitable for serial running and produce output files for entire systems. Unlike the routines listed above, these omit any domain boundary lattice points used in calculations.

Notes regarding `.q`, `.vtk` and `.vts` files

- `lbout`x`.*` is the `.q`, `.vtk` or `.vts` file at the xth saved step during serial running.
- `lbout`y`at`x`.*` is the `.q`, `.vtk` or `.vts` file at the xth saved step written by processor y during parallel running.
- `lbout` files produced using multiple processors will require gathering or simultaneous plotting: see Appendix B for more details.

6.2.9 lbpBOUND

fNextStep

- **Header records** (three cases: 3D, 2D and coordinate)
  `long fNextStep(int q, int xpos, int ypos, int zpos)`
  `long fNextStep(int q, int xpos, int ypos)`
  `long fNextStep(int dx, int dy, int dz, long tpos)`
- **Function**
  Finds particle position at the next time step when currently at (`xpos`, `ypos`, `zpos`) (or `tpos`) and moving along direction `q` or (`dx`, `dy`, `dz`).

- **Dependencies**
  `fCppMod`
- **Arguments**

`q`	input	integer
`dx`	input	integer
`dy`	input	integer
`dz`	input	integer
`xpos`	input	integer
`ypos`	input	integer
`zpos`	input	integer
`tpos`	input	long integer
`fNextStep`	output	long integer

fBounceBackF

- **Header records**
  `int fBounceBackF(long tpos)`
- **Function**
  Performs an on-grid bounce-back for the fluid distribution function at the `tpos`-th grid point.
- **Dependencies**
  None
- **Arguments**

`tpos`	input	long integer

- **Comments**
  This bounce-back boundary condition is carried out using `f(lbopv[i]) = f(lbv[i])`, i.e. populations are exchanged with conjugate values.

fBounceBackC

- **Header records**
  `int fBounceBackC(long tpos)`
- **Function**
  Performs an on-grid bounce-back for the solute distribution function at the `tpos`-th grid point.
- **Dependencies**
  None
- **Arguments**

`tpos`	input	long integer

- **Comments**
  This bounce-back boundary condition is carried out using `f(lbopv[i]) = f(lbv[i])`.

fBounceBackT

- **Header records**
  `int fBounceBackC(long tpos)`
- **Function**
  Performs an on-grid bounce-back for the temperature distribution function at the `tpos`-th grid point.

- **Dependencies**
  None
- **Arguments**
  `tpos` input long integer
- **Comments**
  This bounce-back boundary condition is carried out using `f(lbopv[i]) = f(lbv[i])`.

fMidBounceBackF

- **Header records**
  `int fMidBounceBackF(long tpos)`
- **Function**
  Performs a mid-link bounce-back for the fluid distribution function at the `tpos`-th grid point.
- **Dependencies**
  None
- **Arguments**
  `tpos` input long integer
- **Comments**
  This bounce-back boundary condition is carried out by exchanging post-collisional populations with conjugate values in neighbouring grid points.

fMidBounceBackC

- **Header records**
  `int fMidBounceBackC(long tpos)`
- **Function**
  Performs a mid-link bounce-back for the solute distribution function at the `tpos`-th grid point.
- **Dependencies**
  None
- **Arguments**
  `tpos` input long integer
- **Comments**
  This bounce-back boundary condition is carried out by exchanging post-collisional populations with conjugate values in neighbouring grid points.

fMidBounceBackT

- **Header records**
  `int fMidBounceBackC(long tpos)`
- **Function**
  Performs a mid-link bounce-back for the temperature distribution function at the `tpos`-th grid point.
- **Dependencies**
  None

- **Arguments**
  `tpos` input long integer
- **Comments**
  This bounce-back boundary condition is carried out by exchanging post-collisional populations with conjugate values in neighbouring grid points.

fSiteBlankF

- **Header records**
  `int fSiteBlankF(long tpos)`
- **Function**
  Sets the fluid particle distribution function at the `tpos`-th grid point to zero.
- **Dependencies**
  None
- **Arguments**
  `tpos` input long integer
- **Comments**
  This routine is used to ensure e.g. flows inside solid boundaries are negligible.

fSiteBlankC

- **Header records**
  `int fSiteBlankC(long tpos)`
- **Function**
  Sets the solute particle distribution function at the `tpos`-th grid point to zero.
- **Dependencies**
  None
- **Arguments**
  `tpos` input long integer
- **Comments**
  This routine is used to ensure e.g. diffusion inside a bulk solid is negligible compared to that in a liquid.

fSiteBlankT

- **Header records**
  `int fSiteBlankT(long tpos)`
- **Function**
  Sets the temperature particle distribution function at the `tpos`-th grid point to zero.
- **Dependencies**
  None
- **Arguments**
  `tpos` input long integer
- **Comments**
  This routine is used to ensure e.g. negligible heat transfer through an insulator.

fFixedSpeedFluid

- **Header records**
  `int fFixedSpeedFluid(long tpos, int prop, double *uwall)`
- **Function**
  Calculates the particle distribution function at a fixed speed boundary for compressible fluids.
- **Dependencies**
  Many for different lattice schemes
- **Arguments**

`tpos`	input	long integer
`prop`	input	long integer
`uwall`	output	array of doubles

- **Comments**
  Planar surface calculations are based on [60]; concave edges and corners use equilibrium boundary conditions with the density on the edge and at the grid point assumed to be equal to the values at their nearest neighbours in the bulk fluid. The array `uwall` is the velocity at the grid point for all fluids, which is subsequently used for solute concentration and temperature boundary conditions.

fFixedSpeedFluidIncom

- **Header records**
  `int fFixedSpeedFluidIncom(long tpos, int prop, double *uwall)`
- **Function**
  Calculates the particle distribution function at a fixed speed boundary for incompressible fluids.
- **Dependencies**
  Many for different lattice schemes
- **Arguments**

`tpos`	input	long integer
`prop`	input	long integer
`uwall`	output	array of doubles

- **Comments**
  Planar surface calculations are based on [60]; concave edges and corners use equilibrium boundary conditions with the density on the edge and at the grid point assumed to be equal to the values at their nearest neighbours in the bulk fluid. The array `uwall` is the velocity at the grid point for all fluids, which is subsequently used for solute concentration and temperature boundary conditions.

fFixedDensityFluid

- **Header records**
  `int fFixedDensityFluid(long tpos, int prop, double *uwall)`
- **Function**
  Calculates the particle distribution function at a fixed density boundary for compressible fluids.
- **Dependencies**
  Many for different lattice schemes

- **Arguments**

`tpos`	input	long integer
`prop`	input	long integer
`uwall`	output	array of doubles

- **Comments**
  Planar surface calculations are based on [60]; concave edges and corners assume zero speed at the boundary. The array `uwall` is the velocity at the grid point for all fluids, which is subsequently used for solute concentration and temperature boundary conditions.

fFixedDensityFluidIncom

- **Header records**
  `int fFixedDensityFluidIncom(long tpos, int prop, double *uwall)`

- **Function**
  Calculates the particle distribution function at a fixed density boundary for incompressible fluids.

- **Dependencies**
  Many for different lattice schemes

- **Arguments**

`tpos`	input	long integer
`prop`	input	long integer
`uwall`	output	array of doubles

- **Comments**
  Planar surface calculations are based on [60]; concave edges and corners assume zero speed at the boundary. The array `uwall` is the velocity at the grid point for all fluids, which is subsequently used for solute concentration and temperature boundary conditions.

fFixedSoluteConcen

- **Header records**
  `int fFixedSoluteConcen(long tpos, int prop, double *uwall)`

- **Function**
  Calculates the particle distribution function at a fixed composition boundary for compressible and incompressible fluids.

- **Dependencies**
  Many for different lattice schemes

- **Arguments**

`tpos`	input	long integer
`prop`	input	long integer
`uwall`	input	array of doubles

- **Comments**
  Planar surface calculations are based on [21]; concave edges and corners assume zero speed at the boundary. The fluid velocity at the lattice point, given by the array `uwall`, is required for this boundary condition.

fFixedTemperature

- **Header records**
  `int fFixedTemperature(long tpos, int prop, double *uwall)`
- **Function**
  Calculates the particle distribution function at a fixed temperature boundary for compressible and incompressible fluids.
- **Dependencies**
  Many for different lattice schemes
- **Arguments**

`tpos`	input	long integer
`prop`	input	long integer
`uwall`	input	array of doubles

- **Comments**
  Planar surface calculations are based on [21]; concave edges and corners assume zero speed at the boundary. The fluid velocity at the lattice point, given by the array `uwall`, is required for this boundary condition.

fPostCollBoundary

- **Header records**
  `int fPostCollBoundary()`
- **Function**
  Calculates the particle distribution function at different boundaries after the collision step, prior to propagation, for compressible fluids.
- **Dependencies**
  Many for different lattice schemes
- **Comments**
  Algorithms for other boundary conditions can be added by the user, although care should be taken as to when they are applied in each time step: the conditions invoked in this routine are applied to post-collisional distribution functions before propagation takes place.

fPostCollBoundaryIncom

- **Header records**
  `int fPostCollBoundaryIncom()`
- **Function**
  Calculates the particle distribution function at different boundaries after the collision step, prior to propagation, for incompressible fluids.
- **Dependencies**
  Many for different lattice schemes
- **Comments**
  Algorithms for other boundary conditions can be added by the user, although care should be taken as to when they are applied in each time step: the conditions invoked in this routine are applied to post-collisional distribution functions before propagation takes place.

fPostPropBoundary

- **Header records**
  `int fPostPropBoundary()`

- **Function**
  Calculates the particle distribution function at different boundaries after propagation, prior to the next collision step, for compressible fluids.

- **Dependencies**
  Many for different lattice schemes

- **Comments**
  Algorithms for other boundary conditions can be added by the user, although care should be taken as to when they are applied in each time step: the conditions invoked in this routine apply to distribution functions after propagation.

fPostPropBoundaryIncom

- **Header records**
  `int fPostPropBoundaryIncom()`

- **Function**
  Calculates the particle distribution function at different boundaries after propagation, prior to the next collision step, for incompressible fluids.

- **Dependencies**
  Many for different lattice schemes

- **Comments**
  Algorithms for other boundary conditions can be added by the user, although care should be taken as to when they are applied in each time step: the conditions invoked in this routine apply to distribution functions after propagation.

fsPeriodic

- **Header records**
  `int fsPeriodic()`

- **Function**
  Applies periodic boundary condition for serial calculations with non-zero boundary domain widths by copying distribution functions from edges of fluid points.

- **Dependencies**
  `fsPeriodic2D`
  `fsPeriodic3D`

- **Comments**
  Serial equivalent of `fNonBlockCommunication`, essential for using the combined swap propagation routine `fPropagationCombSwap` in serial running.

fsBoundPeriodic

- **Header records**
  `int fsPeriodic()`

- **Function**
  Applies periodic boundary condition for serial calculations with non-zero boundary domain widths by copying boundary information from edges of fluid points.

- **Dependencies**
  `fsPeriodic2D`
  `fsPeriodic3D`

- **Comments**
  Serial equivalent of `fBoundNonBlockCommunication`, may be required for using combined collision/propagation routines `fCollision*Propagation` in serial running.

fsForcePeriodic

- **Header records**
  `int fsPeriodic()`

- **Function**
  Applies periodic boundary condition for serial calculations with non-zero boundary domain widths by copying interaction forces from edges of fluid points.

- **Dependencies**
  `fsPeriodic2D`
  `fsPeriodic3D`

- **Comments**
  Serial equivalent of `fForceNonBlockCommunication`, may be required for using combined collision/propagation routines `fCollision*Propagation` in serial running.

6.2.10 lbpSUB

fWeakMemory

- **Header records**
  `inline void fWeakMemory()`

- **Function**
  Terminates calculation if system has insufficient memory.

- **Dependencies**
  None

- **Comments**
  If called, will print error message:
  `error: cannot allocate more memory`.

fMemoryAllocation

- **Header records**
  `int fMemoryAllocation()`
- **Function**
  Allocates memory for lattice Boltzmann calculations.
- **Dependencies**
  `fWeakMemory`
- **Comments**
  If memory allocation is unsuccessful, will print error message and stop calculation.

fFreeMemory

- **Header records**
  `int fFreeMemory()`
- **Function**
  Frees allocated memory.
- **Dependencies**
  None

fSetSerialDomain

- **Header records**
  `int fSetSerialDomain()`
- **Function**
  Sets domain parameters for serial running.
- **Dependencies**
  None
- **Comments**
  Default routine, sets domain boundary width `lbdm.bwid` to zero.

fSetSerialDomainBuffer

- **Header records**
  `int fSetSerialDomainBuffer()`
- **Function**
  Sets domain parameters for serial running including additional boundary points.
- **Dependencies**
  None
- **Comments**
  Similar to `fSetSerialDomain` but does not modify the domain boundary width from its user-specified value.

fStartDLMESO

- **Header records**
  `int fStartDLMESO()`
- **Function**
  Announces start of DL_MESO_LBE run.
- **Dependencies**
  None
- **Comments**
  If preferred, the call to this routine can be commented out.

fFinishDLMESO

- **Header records**
  `int fFinishDLMESO()`
- **Function**
  Announces end of DL_MESO_LBE run, prints simulation time and efficiency measure (Millions of Lattice Updates Per Second).
- **Dependencies**
  None
- **Comments**
  If preferred, the call to this routine can be commented out.

fGetModel

- **Header records**
  `int fGetModel()`
- **Function**
  Initializes vectors `lbv`, `lbw` and `lbopv` for lattice model.
- **Dependencies**
  `D2Q9`
  `D3Q15`
  `D3Q19`
  `D3Q27`
- **Comments**
  Parameters are specified according to requested space dimension and number of discrete velocities.

fMarkBoundArea

- **Header records**
  `int fMarkBoundArea()`
- **Function**
  Denotes where boundary areas for message passing and/or periodic boundary conditions are located.
- **Dependencies**
  `int fMarkBoundArea3D()`
  `int fMarkBoundArea2D()`

- **Comments**
  Only used when boundary areas are used (primarily for parallel computing).

fGetEquilibriumF

- **Header records**
  `int fGetEquilibriumF(double *feq, double *v, double rho)`

- **Function**
  Calculates equilibrium distribution function for compressible fluid.

- **Dependencies**
  None

- **Arguments**

`feq`	output	double pointer
`v`	input	double pointer
`rho`	input	double precision

- **Comments**
  The equilibrium distribution function calculated here is

$$f^{eq} = w_i \rho \left[1 + \frac{3\left(\vec{e}_i \cdot \vec{u}\right)}{c^2} + \frac{9\left(\vec{e}_i \cdot \vec{u}\right)^2}{2c^4} - \frac{3u^2}{2c^2} \right]$$

  which is only suitable for square lattices. Other lattice models, e.g. FHP[9], would require modification or alternative versions of this routine.

fGetEquilibriumC

- **Header records**
  `int fGetEquilibriumC(double *feq, double *v, double rho)`

- **Function**
  Calculates equilibrium distribution function for solute.

- **Dependencies**
  None

- **Arguments**

`feq`	output	double pointer
`v`	input	double pointer
`rho`	input	double precision

- **Comments**
  The equilibrium distribution function calculated here is

$$g^{eq} = w_i C \left[1 + \frac{3\left(\vec{e}_i \cdot \vec{u}\right)}{c^2} \right]$$

  using the solute concentration C and the velocity of the bulk fluid[20]. This subroutine can be changed for other Lattice Boltzmann models, e.g. free energy model[51].

fGetEquilibriumT

- **Header records**
  `int fGetEquilibriumT(double *feq, double *v, double tem)`
- **Function**
  Calculates equilibrium distribution function for temperature.
- **Dependencies**
  None
- **Arguments**

`feq`	output	double pointer
`v`	input	double pointer
`tem`	input	double precision

- **Comments**
  The equilibrium distribution function calculated here is

$$h^{eq} = w_i T \left[1 + \frac{3\left(\vec{e}_i \cdot \vec{u}\right)}{c^2}\right]$$

  using the velocity of the bulk fluid[20]. This subroutine can be changed for other Lattice Boltzmann models, e.g. [16].

fGetEquilibriumIncom

- **Header records**
  `int fGetEquilibriumIncom(double *feq, double *v, double rho, double rho0)`
- **Function**
  Calculates equilibrium distribution function for incompressible fluid.
- **Dependencies**
  None
- **Arguments**

`feq`	output	double pointer
`v`	input	double pointer
`rho`	input	double precision
`rho0`	input	double precision

- **Comments**
  Equilibrium distribution function calculated here is

$$f^{eq} = w_i \left\{\rho + \rho_0 \left[\frac{3\left(e_i \cdot u\right)}{c^2} + \frac{9\left(e_i \cdot u\right)^2}{2c^4} - \frac{3u^2}{2c^2}\right]\right\},$$

  which is only suitable for square lattices. Further details can be found in [17].

fGetMomentEquilibriumF

- **Header records**
  `int fGetMomentEquilibriumF(double *meq, double *p, double rho)`
- **Function**
  Calculates equilibrium distribution function in moment space for compressible fluid.

- **Dependencies**
  None
- **Arguments**

`meq`	output	double pointer
`p`	input	double pointer
`rho`	input	double precision

- **Comments**
  The equilibrium distribution function in moment space calculated here is

$$\vec{M}^{eq} = \mathbf{T}\vec{f}^{eq}$$

  the exact form of which is dependent on the lattice scheme; given for D2Q9 by [26] and for D3Q15 and D3Q19 by [7]. Parameters for calculating the square of energy (ϵ) and fourth-order moments ($\pi_{\alpha\alpha}$) can be modified by the user in the `lbpMODEL` module.

fGetMomentEquilibriumIncom

- **Header records**
  `int fGetMomentEquilibriumIncom(double *meq, double *p, double rho, double rho0)`
- **Function**
  Calculates equilibrium distribution function in moment space for incompressible fluid.
- **Dependencies**
  None
- **Arguments**

`meq`	output	double pointer
`p`	input	double pointer
`rho`	input	double precision
`rho0`	input	double precision

- **Comments**
  The equilibrium distribution function in moment space calculated here is

$$\vec{M}^{eq} = \mathbf{T}\vec{f}^{eq}$$

  the exact form of which is dependent on the lattice scheme; given for D2Q9 by [26] and for D3Q15 and D3Q19 by [7]. Parameters for calculating the square of energy (ϵ) and fourth-order moments ($\pi_{\alpha\alpha}$) can be modified by the user in the `lbpMODEL` module.

fGetMomentForce

- **Header records**
  `int fGetMomentForce(double *source, double *v, double *force)`
- **Function**
  Calculates Guo-like forcing terms in moment space.
- **Dependencies**
  None
- **Arguments**

`meq`	output	double pointer
`v`	input	double pointer
`force`	input	double pointer

- **Comments**
  The forcing terms in moment space[39] calculated here are

$$\vec{S} = \mathbf{T}\vec{w} \cdot [3\,(\hat{e}_i - \vec{v}) + 9\,(\hat{e}_i \cdot \vec{v})\,\hat{e}_i] \cdot \vec{F}$$

  the exact form of which is dependent on the lattice scheme (given for D2Q9, D3Q15 and D3Q19).

fGetMRTCollide

- **Header records**
  `int fGetMRTCollide(double *collide, double omegashear, double omegabulk)`
- **Function**
  Calculates collision vector for Multiple-Relaxation-Time (MRT) scheme with specified fluid relaxation frequencies.
- **Dependencies**
  None
- **Arguments**

`collide`	output	double pointer
`omegashear`	input	double precision
`omegabulk`	input	double precision

- **Comments**
  The exact form of the collision vector $\vec{s}$ is dependent on the lattice scheme; given for D2Q9 by [26] and for D3Q15 and D3Q19 by [7]. Tuneable parameters for calculation stability can be modified by the user in the `lbpMODEL` module.

fInitializeSystem

- **Header records**
  `int fInitializeSystem()`
- **Function**
  Initializes distribution function for lattice system using compressible fluids.
- **Dependencies**
  `fGetEquilibriumF`
  `fGetEquilibriumC`
  `fGetEquilibriumT`
- **Comments**
  This subroutine as it stands is suitable for initializing most Lattice Boltzmann systems with compressible fluids, although the user may modify it if it can otherwise be faster, more stable or more suitable for a particular calculation.

fInitializeSystemIncom

- **Header records**
  `int fInitializeSystem()`
- **Function**
  Initializes distribution function for lattice system using incompressible fluids.

- **Dependencies**
  `fGetEquilibriumIncom`
  `fGetEquilibriumC`
  `fGetEquilibriumT`

- **Comments**
  This subroutine as it stands is suitable for initializing most Lattice Boltzmann systems with incompressible fluids, although the user may modify it if it can otherwise be faster, more stable or more suitable for a particular calculation.

fSiteCollisionBGK

- **Header records**
  `int fSiteCollisionBGK(double* startpos, double* bodyforce)`

- **Function**
  Calculates collisions at a grid point using the Bhatnagar-Grook-Krook (BGK) model for compressible fluids.

- **Dependencies**
  `fGetOneMassSite`
  `fGetSpeedSite`
  `fGetEquilibriumF`
  `fGetEquilibriumC`
  `fGetEquilibriumT`

- **Arguments**

`startpos`	input	double pointer
`bodyforce`	input	double pointer

- **Comments**
  Collisions are carried out using the BGK single relaxation time[4]:

$$(\partial_t + e_{i\alpha}\partial_\alpha)\, f_i = -\frac{1}{\tau_f}\,(f_i - f_i^{eq})$$

fSiteCollisionBGKIncom

- **Header records**
  `int fSiteCollisionBGKIncom(double* startpos, double* bodyforce)`

- **Function**
  Calculates collisions at a grid point using the Bhatnagar-Grook-Krook (BGK) model for incompressible fluids.

- **Dependencies**
  `fGetOneMassSite`
  `fGetSpeedIncomSite`
  `fGetEquilibriumIncom`
  `fGetEquilibriumC`
  `fGetEquilibriumT`

- **Arguments**

`startpos`	input	double pointer
`bodyforce`	input	double pointer

- **Comments**
  Collisions are carried out using the BGK single relaxation time[4]:

$$(\partial_t + e_{i\alpha}\partial_\alpha)\, f_i = -\frac{1}{\tau_f}\,(f_i - f_i^{eq})$$

fSiteCollisionBGKGuo

- **Header records**
  `int fSiteCollisionBGKGuo(double* startpos, double* bodyforce)`

- **Function**
  Calculates collisions at a grid point using the Bhatnagar-Grook-Krook (BGK) model with the Guo forcing term for compressible fluids.

- **Dependencies**
  `fGetOneMassSite`
  `fGetSpeedSite`
  `fGetEquilibriumF`
  `fGetEquilibriumC`
  `fGetEquilibriumT`

- **Arguments**

`startpos`	input	double pointer
`bodyforce`	input	double pointer

- **Comments**
  Collisions are carried out using the BGK single relaxation time[4] and the Guo forcing term[14]:

$$(\partial_t + e_{i\alpha}\partial_\alpha)\, f_i = -\frac{1}{\tau_f}\,(f_i - f_i^{eq}) + \left(1 - \frac{1}{2\tau_f}\right) w_i\,[3\,(e_{i\alpha} - v_\alpha) + 9\,(\hat{e}_i \cdot \vec{v})]\,F_\alpha$$

fSiteCollisionBGKGuoIncom

- **Header records**
  `int fSiteCollisionBGKGuoIncom(double* startpos, double* bodyforce)`

- **Function**
  Calculates collisions at a grid point using the Bhatnagar-Grook-Krook (BGK) model with the Guo forcing term for incompressible fluids.

- **Dependencies**
  `fGetOneMassSite`
  `fGetSpeedIncomSite`
  `fGetEquilibriumIncom`
  `fGetEquilibriumC`
  `fGetEquilibriumT`

- **Arguments**

`startpos`	input	double pointer
`bodyforce`	input	double pointer

- **Comments**
  Collisions are carried out using the BGK single relaxation time[4] and the Guo forcing term[14]:

$$(\partial_t + e_{i\alpha}\partial_\alpha)\, f_i = -\frac{1}{\tau_f}\,(f_i - f_i^{eq}) + \left(1 - \frac{1}{2\tau_f}\right) w_i\,[3\,(e_{i\alpha} - v_\alpha) + 9\,(\hat{e}_i \cdot \vec{v})]\,F_\alpha$$

fSiteCollisionMRT

- **Header records**
  `int fSiteCollisionMRT(double* startpos, double* bodyforce)`
- **Function**
  Calculates collisions at a grid point using Multiple-Relaxation-Time (MRT) models for compressible fluids.
- **Dependencies**
  `fGetOneMassSite`
  `fGetSpeedSite`
  `fGetMomentEquilibriumF`
  `fGetMRTCollide`
  `fGetEquilibriumC`
  `fGetEquilibriumT`
- **Arguments**

`startpos`	input	double pointer
`bodyforce`	input	double pointer

- **Comments**
  Collisions are carried out using multiple-relaxation-time (MRT) schemes[26, 7]:

$$(\partial_t + e_{i\alpha}\partial_\alpha) f_i = \mathbf{T}^{-1}\left[-\vec{s}\left(\mathbf{T}\vec{f} - \vec{M}_i^{eq}\right)\right]$$

fSiteCollisionMRTIncom

- **Header records**
  `int fSiteCollisionMRTIncom(double* startpos, double* bodyforce)`
- **Function**
  Calculates collisions at a grid point using Multiple-Relaxation-Time (MRT) models for incompressible fluids.
- **Dependencies**
  `fGetOneMassSite`
  `fGetSpeedIncomSite`
  `fGetMomentEquilibriumIncom`
  `fGetMRTCollide`
  `fGetEquilibriumC`
  `fGetEquilibriumT`
- **Arguments**

`startpos`	input	double pointer
`bodyforce`	input	double pointer

- **Comments**
  Collisions are carried out using multiple-relaxation-time (MRT) schemes[26, 7]:

$$(\partial_t + e_{i\alpha}\partial_\alpha) f_i = \mathbf{T}^{-1}\left[-\vec{s}\left(\mathbf{T}\vec{f} - \vec{M}_i^{eq}\right)\right]$$

fSiteCollisionMRTGuo

- **Header records**
  `int fSiteCollisionMRTGuo(double* startpos, double* bodyforce)`

- **Function**
  Calculates collisions at a grid point using Multiple-Relaxation-Time (MRT) models coupled with Guo-like forcing terms for compressible fluids.

- **Dependencies**
  `fGetOneMassSite`
  `fGetSpeedSite`
  `fGetMomentEquilibriumF`
  `fGetMomentForce`
  `fGetMRTCollide`
  `fGetEquilibriumC`
  `fGetEquilibriumT`

- **Arguments**

`startpos`	input	double pointer
`bodyforce`	input	double pointer

- **Comments**
  Collisions are carried out using multiple-relaxation-time (MRT) schemes coupled with Guo-like forcing terms[39]:

$$\left(\partial_t + e_{i\alpha}\partial_\alpha\right) f_i = \mathbf{T}^{-1}\left[-\vec{s}\left(\mathbf{T}\vec{f} - \vec{M}_i^{eq}\right) + \left(\mathbf{I} - \tfrac{1}{2}\mathbf{I}\vec{s}\right)\vec{S}\right]$$

fSiteCollisionMRTGuoIncom

- **Header records**
  `int fSiteCollisionMRTGuoIncom(double* startpos, double* bodyforce)`

- **Function**
  Calculates collisions at a grid point using Multiple-Relaxation-Time (MRT) models coupled with Guo-like forcing terms for incompressible fluids.

- **Dependencies**
  `fGetOneMassSite`
  `fGetSpeedIncomSite`
  `fGetMomentEquilibriumIncom`
  `fGetMomentForce`
  `fGetMRTCollide`
  `fGetEquilibriumC`
  `fGetEquilibriumT`

- **Arguments**

`startpos`	input	double pointer
`bodyforce`	input	double pointer

- **Comments**
  Collisions are carried out using multiple-relaxation-time (MRT) schemes coupled with Guo-like forcing terms[39]:

$$\left(\partial_t + e_{i\alpha}\partial_\alpha\right) f_i = \mathbf{T}^{-1}\left[-\vec{s}\left(\mathbf{T}\vec{f} - \vec{M}_i^{eq}\right) + \left(\mathbf{I} - \tfrac{1}{2}\mathbf{I}\vec{s}\right)\vec{S}\right]$$

fCalcPotential_ShanChen

- **Header records**
  `int fCalcPotential_ShanChen()`

- **Function**
  Calculates the interaction potential $\phi = 1 - \exp(-\rho)$ as suggested by the Shan-Chen model.
- **Dependencies**
  `fGetOneMassSite`
- **Comments**
  The Shan-Chen model is detailed in [43, 44]. Alternative mesoscale interaction potentials can be introduced here.

fCalcInteraction_ShanChen

- **Header records**
  `int fCalcInteraction_ShanChen(int xpos, int ypos, int zpos)`
- **Function**
  Calculates particle interaction forces according to Shan-Chen model.
- **Arguments**

`xpos`	input	integer
`ypos`	input	integer
`zpos`	input	integer

- **Comments**
  Further details can be found in [43, 44]. Similarly named routines to calculate interaction forces for alternative mesoscale algorithms can be added by the user.

fInteractionForceShanChen

- **Header records**
  `int fInteractionForceShanChen()`
- **Function**
  Calculates interaction forces for all fluids based on the Shan-Chen model[43, 44].
- **Dependencies**
  `fCalcPotential_ShanChen`
  `fCalcInteraction_ShanChen`
- **Comments**
  Alternative mesoscale interactions can be applied using similar routines.

fCollisionBGK

- **Header records**
  `int fCollisionBGK()`
- **Function**
  Collision steps for all compressible fluids using BGK model.
- **Dependencies**
  `fSiteCollisionBGK`
- **Comments**
  This routine is fundamental to Lattice Boltzmann calculations and should not be modified.

fCollisionBGKIncom

- **Header records**
  `int fCollisionBGKIncom()`
- **Function**
  Collision steps for all incompressible fluids using BGK model.
- **Dependencies**
  `fSiteCollisionBGKIncom`
- **Comments**
  This routine is fundamental to Lattice Boltzmann calculations and should not be modified.

fCollisionBGKGuo

- **Header records**
  `int fCollisionBGKGuo()`
- **Function**
  Collision steps for all compressible fluids using BGK model with Guo forcing terms.
- **Dependencies**
  `fSiteCollisionBGKGuo`
- **Comments**
  This routine is fundamental to Lattice Boltzmann calculations and should not be modified.

fCollisionBGKGuoIncom

- **Header records**
  `int fCollisionBGKGuoIncom()`
- **Function**
  Collision steps for all incompressible fluids using BGK model with Guo forcing terms.
- **Dependencies**
  `fSiteCollisionBGKGuoIncom`
- **Comments**
  This routine is fundamental to Lattice Boltzmann calculations and should not be modified.

fCollisionMRT

- **Header records**
  `int fCollisionMRT()`
- **Function**
  Collision steps for all compressible fluids using MRT model.
- **Dependencies**
  `fSiteCollisionMRT`
- **Comments**
  This routine is fundamental to Lattice Boltzmann calculations and should not be modified.

fCollisionMRTIncom

- **Header records**
  `int fCollisionMRTIncom()`
- **Function**
  Collision steps for all incompressible fluids using MRT model.
- **Dependencies**
  `fSiteCollisionMRTIncom`
- **Comments**
  This routine is fundamental to Lattice Boltzmann calculations and should not be modified.

fCollisionMRTGuo

- **Header records**
  `int fCollisionMRTGuo()`
- **Function**
  Collision steps for all compressible fluids using MRT model with Guo-like forcing terms.
- **Dependencies**
  `fSiteCollisionMRTGuo`
- **Comments**
  This routine is fundamental to Lattice Boltzmann calculations and should not be modified.

fCollisionMRTGuoIncom

- **Header records**
  `int fCollisionMRTGuoIncom()`
- **Function**
  Collision steps for all incompressible fluids using MRT model with Guo-like forcing terms.
- **Dependencies**
  `fSiteCollisionMRTGuoIncom`
- **Comments**
  This routine is fundamental to Lattice Boltzmann calculations and should not be modified.

fPropagation

- **Header records**
  `int fPropagation()`
- **Function**
  Moves lattice particles (distribution functions) to neighbouring grid points using the two-lattice algorithm.
- **Dependencies**
  None
- **Comments**
  This routine is fundamental to Lattice Boltzmann calculations and should not be modified. This is the least efficient propagation routine available.

fPropagationSwap

- **Header records**
  `int fPropagationSwap()`
- **Function**
  Moves lattice particles (distribution functions) to neighbouring grid points using the swap algorithm.
- **Dependencies**
  `fSwapPair`
- **Comments**
  This routine is fundamental to Lattice Boltzmann calculations and should not be modified. Propagation is carried out by systematic swapping of post-collisional values for the distribution function, initially at each grid point and then between them (in two separate loops), as described by [34] and in section 5.1. This version can be used for either serial or parallel calculations and no boundary layer is necessary: this is the default propagation routine for serial calculations.

fPropagationCombSwap

- **Header records**
  `int fPropagationCombSwap()`
- **Function**
  Moves lattice particles (distribution functions) to neighbouring grid points using the swap algorithm.
- **Dependencies**
  `fSwapPair`
- **Comments**
  This routine is fundamental to Lattice Boltzmann calculations and should not be modified. Propagation is carried out by systematic swapping of post-collisional values for the distribution function, initially at each grid point and then between them (in the same loop), as described by [34] and in section 5.1. This version can only be used for calculations with non-zero boundary layers: this is the default propagation routine for parallel calculations.

6.2.11 lbpMPI

This package is only required for parallel running and does not require detailed knowledge for its use. Several subroutines in this package are not described here: interested users should consult the code for further information.

fStartMPI

- **Header records**
  `int fStartMPI(int argc, char* argv[])`
- **Function**
  Starts Message Passing Interface (MPI).
- **Dependencies**
  None

fCloseMPI

- **Header records**
  `int fCloseMPI()`
- **Function**
  Closes Message Passing Interface (MPI).
- **Dependencies**
  None

fGlobalValue

- **Header records** (six cases)
  `int fGlobalValue(double *vqua, int nnum)`
  `int fGlobalValue(int *vqua, int nnum, int *vtot)`
  `int fGlobalValue(int *vqua, int nnum)`
  `int fGlobalValue(long int *vqua, int nnum)`
  `int fGlobalValue(long int *vqua, int nnum, long int *vtot)`
- **Function**
  Sums values from all processes and distributes the sum.

fGlobalProduct

- **Header records** (two cases)
  `int fGlobalProduct(double *vqua, int nnum)`
  `int fGlobalProduct(int *vqua, int nnum)`
- **Function**
  Multiplies together values from all processors and distributes the product.

fArrangeProcessors

- **Header records**
  `int fArrangeProcessors()`
- **Function**
  Arrange processors according to system dimensions.
- **Comments**
  Calculations are based on

$$\frac{\texttt{lbdm.xdim}}{\texttt{lbsy.nx}} \simeq \frac{\texttt{lbdm.ydim}}{\texttt{lbsy.ny}} \simeq \frac{\texttt{lbdm.zdim}}{\texttt{lbsy.nz}}$$

$$\texttt{lbdm.xdim} \times \texttt{lbdm.ydim} \times \texttt{lbdm.zdim} = \texttt{lbdm.size}$$

fDefineDomain

- **Header records**
  `int fDefineDomain()`
- **Function**
  Determines domain parameters for system calculation.

fDefineMessage

- **Header records**
  `int fDefineMessage()`
- **Function**
  Defines vector messages for system (distribution functions, boundary properties and interaction forces).

fDefineNeighbours

- **Header records**
  `int fDefineNeighbours()`
- **Function**
  Calculates the names of neighbouring processes and the start points for sending and receiving messages.
- **Comments**
  This subroutine must *not* be changed!

fNonBlockCommunication

- **Header records**
  `int fNonBlockCommunication()`
- **Function**
  Passes distribution function information for either 2D or 3D system.

fOutputInfo

- **Header records**
  `int fOutputInfo()`
- **Function**
  Outputs number of processes and lengths of integers and floats.
- **Comments**
  This subroutine is necessary for gathering and rearranging the `lbout` data, and produces the files `lbout.info` and `lbout.ext`.

fBoundNonBlockCommunication

- **Header records**
  `int fBoundNonBlockCommunication()`
- **Function**
  Passes boundary information for either 2D or 3D systems.

fForceNonBlockCommunication

- **Header records**
  `int fForceNonBlockCommunication()`
- **Function**
  Passes interaction force information for either 2D or 3D systems.

fCheckTimeMPI

- **Header records**
  `int fCheckTimeMPI()`
- **Function**
  Outputs time in seconds since initial call.
- **Arguments**
  `fCheckTimeMPI` output double
- **Comments**
  Obtains calculation time based on MPI wall clock.

fPrintSystemMass

- **Header records**
  `int fPrintSystemMass()`
- **Function**
  Calculates and prints total and individual fluid masses in entire system.
- **Dependencies**
  `fGetTotMassDomain`
  `fGetOneMassDomain`
  `fGlobalValue`

fPrintSystemMomentum

- **Header records**
  `int fPrintSystemMomentum()`
- **Function**
  Calculates and prints the total fluid momentum in entire system.
- **Dependencies**
  `fGetTotMomentDomain`
  `fGlobalValue`

6.3 DL_MESO_LBE Data Files

6.3.1 Input files

Define system: `lbin.sys`

The use of the DL_MESO GUI is recommended for producing `lbin.sys`, although existing files of that name can also be edited. Superfluous parameters can be deleted and new parameters introduced, although the latter requires an addition to the parameter recognition loop in the `fInputParameters` subroutine.

Define space: `lbin.spa`

The GUI is recommended for creating the `lbin.spa` file, which stores the data in the following format:

`x,y,z,grid property`

An empty `lbin.spa` file represents all boundaries as periodic.

Define initial condition: `lbin.init`

This optional file cannot currently be created by the GUI: the user must create this file if it is required. The following format is required for each lattice point whose default velocity, fluid densities, solute concentrations or temperature needs replacing:

$$\texttt{x,y,z,u}_x\texttt{,u}_y\texttt{,u}_z\texttt{,}\rho_0 \ldots \rho_{\texttt{lbsy.nf}-1}\texttt{,}c_0 \ldots c_{\texttt{lbsy.nc}-1}\texttt{,}T$$

Note that solute concentrations and temperature can be omitted if these are not required in LBE simulations.

6.3.2 Output files

DL_MESO_LBE can write standard Plot3D, Structured Grid Legacy VTK and Structured Grid XML VTK data format files, in binary format for parallel calculations and ANSI for serial. These may be modified by the user as required. The utility `lbeplot3dgather` in the `LBE/utility` directory can combine Plot3D files generated in parallel, while Parallel Structured Grid XML VTK files (`lbtout*.pvts`) that refer to the files from each processor can be created using the utility `lbevtkgather`: further details can be found in Appendix B or the `README` file in the same directory.

Plot3D format

Output grid position: `lbout*.xyz`

$$\begin{aligned}
&nx, ny, nz \\
&x_{0,0,0}, \ldots x_{nx-1,ny-1,nz-1} \\
&y_{0,0,0}, \ldots y_{nx-1,ny-1,nz-1} \\
&z_{0,0,0}, \ldots z_{nx-1,ny-1,nz-1}
\end{aligned}$$

where nx is the total number of grid points in x-direction, ny is the total number of grid points in y-direction, nz is the total number of grid points in z-direction and $(x_{i,j,k}, y_{i,j,k}, z_{i,j,k})$ is the Cartesian coordinate of grid point (i, j, k).

Output macroscopic quantities: `lbout.q`

$$\begin{aligned}
&nx, ny, nz \\
&c, 1.0, Re, t \\
&\rho_{0,0,0}, \ldots \rho_{nx-1,ny-1,nz-1}, U_{0,0,0}, \ldots U_{nx-1,ny-1,nz-1} \\
&V_{0,0,0}, \ldots V_{nx-1,ny-1,nz-1}, W_{0,0,0}, \ldots W_{nx-1,ny-1,nz-1} \\
&\phi_{0,0,0}, \ldots \phi_{nx-1,ny-1,nz-1}
\end{aligned}$$

where c is the speed of sound for the lattice, Re the Reynolds number for the flow, t the time step, ρ the density, U the x-component of velocity, V the y-component of velocity, W the z-component of velocity and ϕ the space (boundary condition) property. (The 1.0 between the lattice speed of sound and flow Reynolds number represents the freestream angle of attack.) The density of the fluid may be replaced with its concentration or scalar temperature.

Legacy VTK format: `lbout*.vtk`

Legacy VTK files written by DL_MESO_LBE include the lattice dimensions (numbers of grid points in each direction), the Cartesian coordinates of the grid points in real-life units, and the following data:

- A scalar property (fluid density, solute concentration or scalar temperature), e.g.

$$\rho_{0,0,0}$$
$$\dots$$
$$\rho_{nx-1,ny-1,nz-1}$$

- Fluid velocity

$$U_{0,0,0}\ V_{0,0,0}\ W_{0,0,0}$$
$$\dots$$
$$U_{nx-1,ny-1,nz-1}\ V_{nx-1,ny-1,nz-1}\ W_{nx-1,ny-1,nz-1}$$

- The space (boundary condition) property

$$\phi_{0,0,0}$$
$$\dots$$
$$\phi_{nx-1,ny-1,nz-1}$$

Structured Grid XML VTK format: `lbout*.vts`

Structured Grid VTK files written by DL_MESO_LBE include the lattice dimensions (numbers of grid points in each direction), the Cartesian coordinates of the grid points in real-life units, and the same data as in Legacy VTK files. Output files produced in serial include the data between XML tags, e.g. `<DataArray>`, while those produced in parallel use the `<DataArray>` tags to refer to the starting point for the data in a stream of binary numbers inside an `<AppendedData>` tag. The latter represent the data in each subdomain and should be retained when plotting the entire system since the parallel VTK format only links to the piece VTK files.

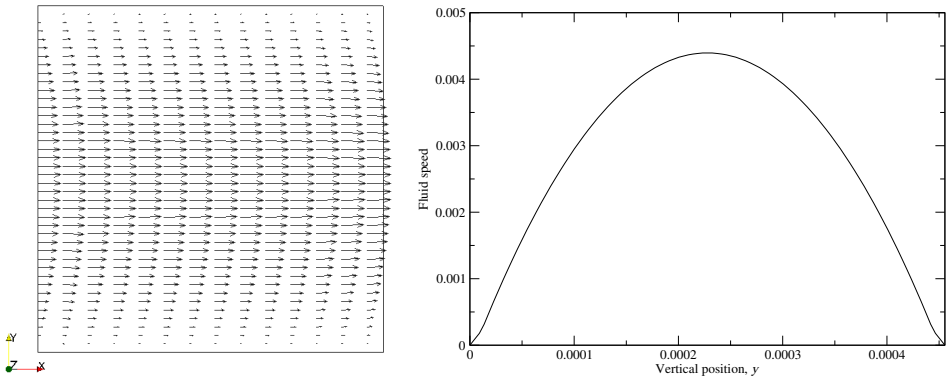
Chapter 7

DL_MESO LBE Examples

Test cases for Lattice Boltzmann Equation simulations using DL_MESO – including the required input and sample output files – can be found in the `DEMO/LBE` subdirectory. They can be run using either the serial or parallel versions of DL_MESO_LBE. They consist of the following cases.

7.1 2D_Pressure

This is a 2D simulation of a single fluid on a 42×42 grid with fixed pressure (density) boundary conditions on the left and right boundaries and solid walls (represented by bounce back) at the top and bottom. A visualization with vector glyphs and a plot of fluid speed against vertical position can be seen in Figure 7.1, which show the boundary conditions result in a laminar flow with a parabolic velocity profile.

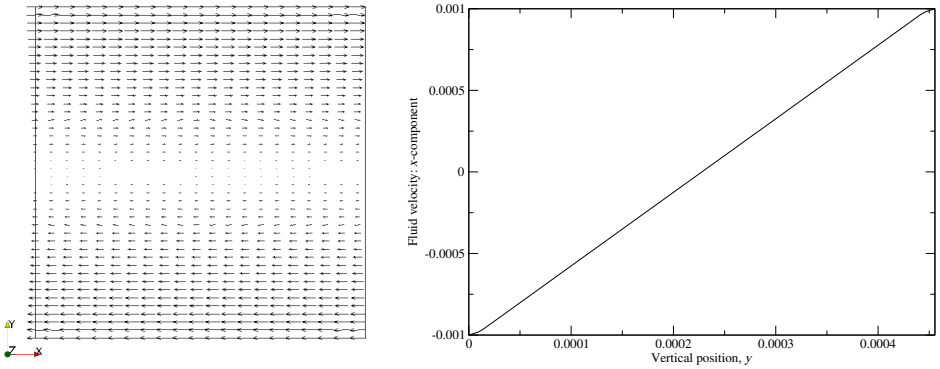


(a) Vector plot of system (b) Variation of fluid speed with vertical position

Figure 7.1: Results from LBE `2D_Pressure` test case

7.2 2D_Shear

This is a 2D simulation of a single fluid on a 42×42 grid with a shear boundary condition. The vector plot in Figure 7.2 demonstrates the ability of the applied boundary conditions to generate a linear shearing Couette flow throughout the grid, which is confirmed by the plot of horizontal velocity component as a function of vertical position at the last time step for the simulation.



(a) Vector plot of system

(b) Variation of horizontal velocity with vertical position

Figure 7.2: Results from LBE `2D_Shear` test case

7.3 2D_CylinderFlow

This is a 2D simulation of a single fluid on a 125×50 grid with a constant horizontal body force and a circular obstacle of radius 12, representing channel flow past an infinitely-long cylinder. Figure 7.3 shows this flow against a map of the system, red representing the fluid and blue indicating solid objects (the walls and cylinder).

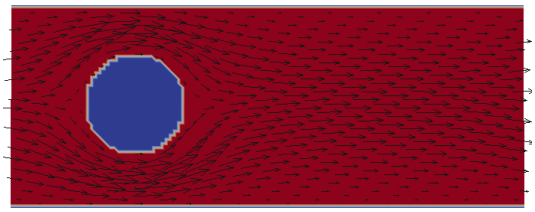


Figure 7.3: Density and velocity vector plot from LBE `2D_CylinderFlow` test case

7.4 2D_KarmanVortex

This is a 2D simulation of a single fluid on a 250×50 grid with a constant horizontal body force and a circular obstacle of radius 8, representing channel flow past an infinitely-long cylinder that eventually produces a von Kármán vortex street between two solid walls. Figure 7.4 shows the flow field at the final time step: an `.AVI` video file has been rendered from the calculation and can be found in the subdirectory for this case.

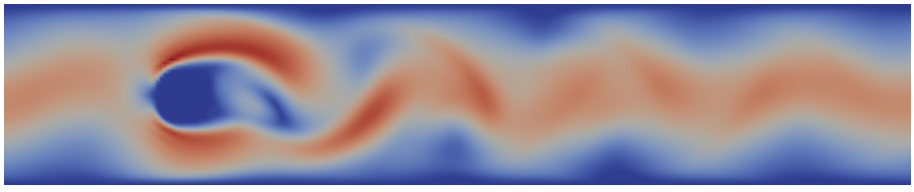


Figure 7.4: Velocity magnitude plot from LBE `2D_KarmanVortex` test case (scale: blue to red)

7.5 2D_LidCavity

This is a 2D simulation of a single fluid on a 128×128 grid with a shear boundary condition at the top and solid walls surrounding the other edges of the system, resulting in lid-driven cavity flow. Figure 7.5 shows the fully-developed velocity field for a Reynolds number of 100 at the final time step.

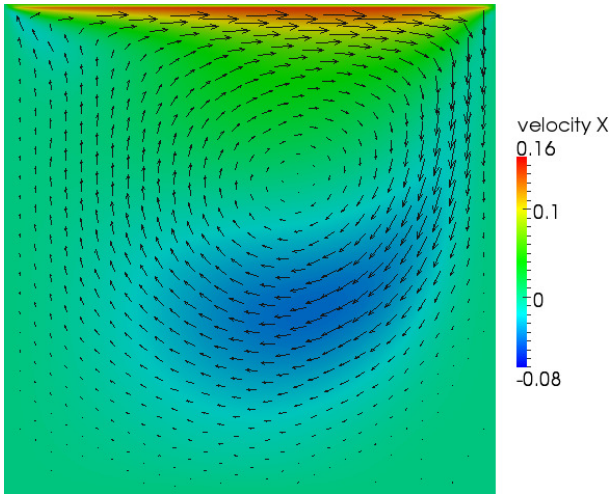
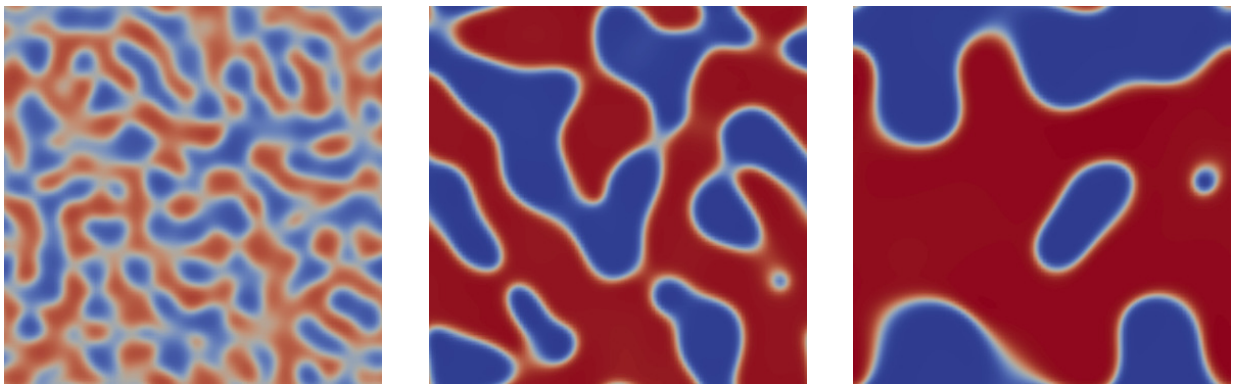


Figure 7.5: Magnitude plot of x-component velocity and velocity vector plot from LBE `2D_Lidcavity` test case

7.6 3D_PhaseSeparation

This is a 3D simulation of a binary fluid on a $100 \times 100 \times 100$ grid with a fixed boundary condition and non-zero mesoscopic interaction parameters, producing phase separation of the two fluids. Figure 7.6 shows the phase separation process in a number of snapshots: two `.AVI` video files have been rendered from an example calculation (one giving a 3D view of the system, the other showing a plane normal to the z-axis) which can be found in the directory for this case.

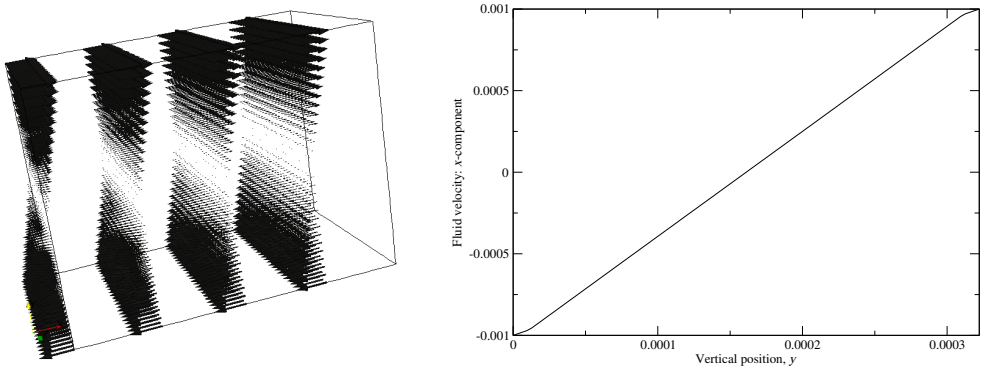


(a) $t = 400$ (b) $t = 1200$ (c) $t = 2000$

Figure 7.6: Progressive density plots in plane normal to z-axis from LBE `3D_PhaseSeparation` test case (red for fluid 0, blue for fluid 1)

7.7 3D_Shear

This is a 3D simulation of a single fluid on a $40 \times 30 \times 25$ grid with a shear boundary condition. Figure 7.7 shows a vector plot for this system, demonstrating that linear shear is generated and maintained by the moving boundaries in the planes normal to the y-axis, and a plot of the horizontal component of fluid velocity against vertical position at the last time step.



(a) Vector plot of system (b) Plot of horizontal velocity with vertical position

Figure 7.7: Results from LBE `3D_Shear` test case

Part II

Dissipative Particle Dynamics (DPD)

Chapter 8

Dissipative Particle Dynamics: Basic Theory

8.1 Introduction

Dissipative Particle Dynamics (DPD) is an off-lattice, discrete particle method for modelling mesoscopic systems. It has little in common with Lattice Boltzmann methods, except in its application to systems of similar length and time scales.

The DPD method inherits its methodology from classical Molecular Dynamics (MD), particularly from Brownian Dynamics (BD). It differs from BD, however, in an important way: it is *Galilean invariant* and for this reason conserves hydrodynamic behaviour, while the BD method does not. Many systems are crucially dependent on hydrodynamic interactions and it is essential to retain this feature in the model. DPD is particularly useful for simulating systems on the near-molecular scale, such as polymers, biopolymers, lipids, emulsions and surfactants – systems in which large scale structure evolves on a time scale that is too long to be modelled effectively by MD. DPD may also be used when such systems experience shear and flow gradients.

The DPD algorithm can be summarized by the following:

- A condensed phase system may be modelled as a system of free particles interacting directly through 'soft' forces.
- The system is coupled to a heat bath via stochastic forces, which act on the particles in a pairwise manner.
- The particles also experience a damping or drag force, which also acts in a pairwise manner.
- Thermodynamic equilibrium is maintained through the balance of the stochastic and drag forces, i.e. the method satisfies the fluctuation-dissipation theorem.
- At equilibrium (or steady state) the properties of the system are calculated as averages over the individual particles, as in Molecular Dynamics.

8.2 Outline of Method

In DPD[1] the system is modelled as a system of free particles, which are spherical and interact over a range that is of the same order as their diameters. The particles can be thought of as assemblies or aggregates of molecules, such as solvent molecules or polymers, or more simply as carriers of momentum.

[1]The outline of the DPD method supplied here is based on [13].

The equations governing the time evolution in a DPD simulation resemble those of ordinary MD:

$$\frac{d\vec{v}_i}{dt} = \frac{\vec{F}_i}{m_i} \tag{8.1}$$

$$\frac{d\vec{r}_i}{dt} = \vec{v}_i \tag{8.2}$$

in which $\vec{r}_i$, $\vec{v}_i$ and $\vec{F}_i$ are the position, velocity and force of the ith particle, which has mass m_i. The force on the particle is a sum of pair forces:

$$\vec{F}_i = \sum_{j \neq i}^{N} \left(\vec{F}_{ij}^C + \vec{F}_{ij}^D + \vec{F}_{ij}^R \right) \tag{8.3}$$

in which $\vec{F}_{ij}^C$, $\vec{F}_{ij}^D$ and $\vec{F}_{ij}^R$ are the *conservative*, *drag* and *random* (or *stochastic*) pair forces respectively. Each represents the force exerted on particle i due to the presence of particle j. Additional pairwise forces may be included for more complicated systems, such as those involving chains of particles bonded together[40].

The conservative interactions are usually 'soft' (i.e. weakly interacting) so that the particles can pass by each other (or even through each other) relatively easily so that equilibrium is achieved quickly. A common form of interaction potential is an inverse parabola:

$$V(r_{ij}) = \frac{1}{2} A_{ij} \left(1 - \frac{r_{ij}}{r_c} \right)^2 \tag{8.4}$$

where $r_{ij} = |\vec{r}_i - \vec{r}_j|$, r_c is a cutoff radius and A_{ij} is the interaction strength. A_{ij} may be the same for all particle pairs or may be different for different particle types.

Equation (8.4) gives rise to a repulsive force of the form

$$\vec{F}_{ij}^C = A_{ij} w^C(r_{ij}) \frac{\vec{r}_{ij}}{r_{ij}} = A_{ij} \left(1 - \frac{r_{ij}}{r_c} \right) \frac{\vec{r}_{ij}}{r_{ij}} \tag{8.5}$$

This is the deterministic or *conservative* force $\vec{F}_i^C$ exerted on particle i by particle j. Note the switching function $w^C(r_{ij})$ and the force are zero when $r_{ij} > r_c$ and thus the particles have an effective diameter of 1 in units of the cutoff radius r_c.

The stochastic forces experienced by the particles is again pairwise in nature and takes the form

$$\vec{F}_{ij}^R = \sigma_{ij} w^R(r_{ij}) \zeta_{ij} \Delta t^{-\frac{1}{2}} \frac{\vec{r}_{ij}}{r_{ij}} \tag{8.6}$$

in which Δt is the time step and $w^R(r_{ij})$ is a switching function which imposes a finite limit on the range of the stochastic force. ζ_{ij} is a random number with zero mean and unit variance. The constant σ_{ij} is related to the temperature, as is understood from the role of the stochastic force in representing a heat bath.

Finally the particles are subject to a drag force, which depends on the relative velocity between interacting pairs of particles:

$$\vec{F}_{ij}^D = -\gamma_{ij} w^D(r_{ij}) (\vec{r}_{ij} \cdot \vec{v}_{ij}) \frac{\vec{r}_{ij}}{r_{ij}^2} \tag{8.7}$$

where $w^D(r_{ij})$ is once again a switching function and $\vec{v_{ij}} = \vec{v_i} - \vec{v_j}$. The constant γ_{ij} is the drag coefficient. It follows from the fluctuation-dissipation theorem that for thermodynamic equilibrium to result from this method the following relations must hold.

$$\sigma_{ij}^2 = 2\gamma_{ij} k_B T \tag{8.8}$$

$$w^D(r_{ij}) = \left[w^R(r_{ij}) \right]^2 \tag{8.9}$$

In practice the switching functions are defined through

$$w^D(r_{ij}) = \left(1 - \frac{r_{ij}}{r_c} \right)^2 \qquad (r_{ij} < r_c) \tag{8.10}$$

which ensures that all interactions are switched off at the range $r_{ij} = r_c$. In many DPD simulations, the stochastic and drag coefficients are often constant for all interactions, i.e. $\sigma_{ij} \equiv \sigma$ and $\gamma_{ij} \equiv \gamma$, although this assumption does not have to apply.

8.3 Equation of state and dynamic properties

The form of the conservative force determines the equation of state for a DPD fluid, which can be derived using the virial theorem to express system pressure as follows:

$$p = \rho k_B T + \frac{1}{3V}\left\langle \sum_{j>i} (\vec{r}_i - \vec{r}_j) \cdot \vec{F}_{ij}^{C} \right\rangle \tag{8.11}$$

$$= \rho k_B T + \frac{2\pi}{3}\rho^2 \int_0^1 rA\left(1 - \frac{r}{r_c}\right) g(r) r^2 \; dr \tag{8.12}$$

where $g(r)$ is a radial distribution function for the soft sphere model[13] and ρ is the DPD particle density. For sufficiently large densities ($\rho > 2$), $g(r)$ takes the same form and the equation of state can be well-approximated by:

$$p = \rho k_B T + \alpha A \rho^2 \tag{8.13}$$

with the parameter $\alpha \approx 0.101 \pm 0.001$. This expression permits the use of fluid compressibilities to obtain conservative force parameters for bulk fluids, e.g. for water $A \approx \frac{75 k_B T}{\rho}$. Alternative equations of state may be obtained by modifying the functional form of conservative interactions to include localized densities (i.e. many-body DPD)[37, 53].

Transport coefficients for a DPD fluid can be derived using the expressions for the drag and stochastic forces[13, 25, 31]. The kinematic viscosity can be found to be

$$\nu \approx \frac{45 k_B T}{4\pi\gamma\rho r_c^3} + \frac{2\pi\gamma\rho r_c^5}{1575} \tag{8.14}$$

while the self-diffusion coefficient is given as

$$D \approx \frac{45 k_B T}{2\pi\gamma\rho r_c^3}. \tag{8.15}$$

The ratio of these two properties, the Schmidt number ($\mathrm{Sc} = \frac{\nu}{D}$), is therefore:

$$\mathrm{Sc} \approx \frac{1}{2} + \frac{(2\pi\gamma\rho r_c^4)^2}{70875 k_B T} \tag{8.16}$$

and for values of the drag coefficient and density frequently used in DPD simulations, this value is of the order of unity, which is an appropriate magnitude for gases but three orders of magnitude too small for liquids.

This property of standard DPD does *not* rule it out for simulations of liquid phases except when hydrodynamics are important. It may also be argued that the self-diffusion of DPD particles might not correspond to that of individual molecules and thus a Schmidt number of the order 10^3 is unnecessary for modelling liquids[38]. Alternative thermostats are available which can model systems with higher Schmidt numbers[27, 49].

8.4 Derivation of Equilibrium

The derivation of the DPD algorithm is based on the Fokker-Planck equation

$$\frac{\partial \rho}{\partial t} = \mathcal{L}\rho \tag{8.17}$$

where ρ is the equilibrium distribution function and $\mathcal{L}$ is the evolution operator, which may be split into *conservative* and *dissipative* parts:

$$\mathcal{L} = \mathcal{L}^{\mathcal{C}} + \mathcal{L}^{\mathcal{D}} \tag{8.18}$$

with

$$\mathcal{L}^{\mathcal{C}} = -\sum_{i=1}^{N} \frac{\vec{p}_i}{m_i} \frac{\partial}{\partial \vec{r}_i} - \sum_{i \neq j}^{N} \vec{F}_{ij}^{C} \frac{\partial}{\partial \vec{p}_i} \tag{8.19}$$

$$\mathcal{L}^{\mathcal{D}} = \sum_{i=1}^{N} \hat{e}_{ij} \cdot \frac{\partial}{\partial \vec{p}_i} \left[\gamma w^D \left(\hat{e}_{ij} \cdot \vec{v}_{ij} \right) + \frac{\sigma^2}{2} \left\{ w^R \left(r_{ij} \right) \right\}^2 \hat{e}_{ij} \cdot \left\{ \frac{\partial}{\partial \vec{p}_i} - \frac{\partial}{\partial \vec{p}_j} \right\} \right] \tag{8.20}$$

where $\hat{e}_{ij} = \frac{\vec{r_{ij}}}{r_{ij}}$.

When $\alpha = \gamma = 0$ then Equation (8.17) becomes

$$\frac{\partial \rho}{\partial t} = \mathcal{L}^{\mathcal{C}} \rho \tag{8.21}$$

for which the equilibrium solution is evidently

$$\rho^{eq} = \frac{1}{Z} \exp \left(\frac{1}{k_B T} \left[\sum_{i=1}^{N} \frac{p_i^2}{2m_i} + \frac{1}{2} \sum_{j \neq i}^{N} \phi \left(r_{ij} \right) \right] \right) \tag{8.22}$$

which is, of course, the Boltzmann distribution function for an equilibrium system. Thus it is apparent that for the simulation based on Equation (8.17) to maintain the same distribution function, the terms in the operator $\mathcal{L}^{\mathcal{D}}$ of Equation (8.20) must sum to zero. It follows that the conditions given in Equations (8.8) and (8.9) must apply.

8.5 Summary of Dissipative Particle Dynamics

DPD is a simple method. All that is required is a system of spherical particles enclosed in a periodic box undergoing time evolution as a result of the above forces. In implementation it differs very little from Molecular Dynamics. It should be noted that all computed interactions are pairwise, which means that the principle of the conservation of momentum in the system, or 'Galilean invariance', is preserved. The conservation of momentum is required for the preservation of hydrodynamic forces.

Chapter 9

DL_MESO_DPD Basic Definition

9.1 Data structure

9.1.1 Storage of running information

DL_MESO_DPD contains storage for information on the system being modelled, the domain and neighbour information for parallel running. The parameters for these aspects of calculations can be found in Tables 9.1, 9.2 and 9.3.

It should be noted for the parameters in Table 9.1 that the fundamental units for the simulation are those of mass $[M]$, length $[L]$ and energy $[E]$: the DPD unit of time is equivalent to $[L]\sqrt{\frac{[M]}{[E]}}$ while temperature (in the form k_BT) is defined as two-thirds of the kinetic energy per particle.

Table 9.1: System information

parameter	meaning
text	name of calculation
volm	(initial) volume of system
dimx, dimy, dimz	(initial) dimensions of system
nsyst	total number of particles
nusyst	total number of unbonded particles
nspe	number of particle species
nmoldef	number of molecule types
temp	specified system temperature (k_BT)
prszero	specified system pressure (P_0)
rcut	interaction cutoff radius (r_c)
rmbcut	many-body interaction cutoff radius (r_d)
relec	short-range electrostatic interaction cutoff radius (r_e)
srfzcut	surface repulsion cutoff distance (z_c)
rhalo	size of boundary halo
nrun	number of calculation timesteps
nseql	number of equilibration timesteps
tstep	duration of calculation timestep (Δt)
timjob	maximum time available to run calculation
tclose	time required to shut down calculation
kres	calculation restart parameter
nsbpo	interval for printing data to OUTPUT file
nsbsv	interval for saving data to HISTORY file(s)
nstk	size of statistical data stack
ltemp	switch for temperature scaling before equilibration
nsbts	interval for temperature scaling
lcorr	switch for saving statistical data to CORREL file
iscorr	interval for saving statistical data to CORREL file
ktype	potential energy function selection
itype	integrator/thermostat selection
btype	barostat selection
etype	electrostatic algorithm selection
srftype	surface boundary selection
lbond	switch for modelling bonds between particles
langle	switch for modelling bond angles
ldihed	switch for modelling bond dihedrals
lgbnd	switch for globally storing all bond data
lisoprs	switch for isotropic variation of system dimensions with pressure

9.1.2 Storage of particle and bond properties

The total number of particles in a system is nsyst, of which nusyst particles are 'loose', i.e. not bonded to other particles. DL_MESO_DPD divides up the particles and total system volume (volm) between the processing units available. At any given time each process holds nbeads particles. Each process also has nbonds bonds, nangles bond angles and ndiheds bond dihedrals to deal with. If bonds are dealt with locally, only the bonds associated with the subdomain are calculated by each process; otherwise all processes hold information on all bonds.

The Cartesian coordinates, velocities and forces for the particles are each held in sets of three double precision arrays for x-, y- and z-components. Particle positions relative to the volume modelled by the individual processor — thus *not* absolute positions unless the serial version of DL_MESO_DPD is used — are held in arrays xxx(i), yyy(i) and zzz(i) (for particle i). Particle velocities are held in vxx(i), vyy(i) and vzz(i). Two sets of arrays for the net forces acting on the particles are available: fxx(i), fyy(i) and fzz(i) for forces that remain constant over each time step, and fvx(i), fvy(i) and fvz(i) for forces that may vary during the time step (e.g. drag forces for DPD Velocity Verlet integration). The potential energy and virial for individual particles

Table 9.2: Domain information

parameter	meaning
idnode	name of the processor
nodes	number of processors
idx	x-coordinate of the processor
idy	y-coordinate of the processor
idz	z-coordinate of the processor
npx	number of processors along x-axis
npy	number of processors along y-axis
npz	number of processors along z-axis
volm	volume of system
dimx	size of system in x-dimension
dimy	size of system in y-dimension
dimz	size of system in z-dimension
sidex	size of domain in x-dimension
sidey	size of domain in y-dimension
sidez	size of domain in z-dimension
delx	absolute x-coordinate of domain origin
dely	absolute y-coordinate of domain origin
delz	absolute z-coordinate of domain origin
nlx	number of link cells in domain along x-axis
nly	number of link cells in domain along y-axis
nlz	number of link cells in domain along z-axis
wdthx	link cell size in x-dimension
wdthy	link cell size in y-dimension
wdthz	link cell size in z-dimension

Table 9.3: Neighbour information

parameter	meaning
map(k)	processor name of neighbour k
k = 1	left neighbour
k = 2	right neighbour
k = 3	lower neighbour
k = 4	upper neighbour
k = 5	back neighbour
k = 6	front neighbour

can also be found in pot(i) and vrl(i).

The particles modelled by a particular processor have both local and global identity numbers, which are stored in the integer arrays loc(i) and lab(i) respectively. Numbers representing species and molecule types are stored in arrays ltp(i) and ltm(i), which are used to assign particle masses and the names of particles and molecules respectively to the arrays weight(i), atmnam(i) and molnam(i).

Bonded particles are listed by global identity numbers in the integer array bndtbl(i,j) for bond i, with j=1 representing the first particle in the pair, j=2 for the second and j=3 giving the user-defined bond type. The location of the first particle in each bond pair determines the processing unit which holds this data; thus movement of this reference particle across processes also causes the bond list entry to be transferred with it. Bond angles and bond dihedrals are stored in similar tables, angtbl(i,j) (particles j from 1 to 3, angle type at j=4) and dhdtbl(i,j) (particles j from 1 to 4, dihedral type at j=5) respectively, using the second particle in each triple or quadruple as the reference particle.

Prior to force calculations, a list of bonded particles in each process domain and the surrounding boundary halo is constructed, lblclst(i,j), to allow DL_MESO_DPD to find the local number for a particle (j=2) from its global number (j=1) using a binary search. This list may include duplicates for the same global particle number; the local numbers giving the shortest distance between pairs of particles are selected and used.

9.2 The Parameters and Their Functions

Table 9.4 lists all the globally used parameters defined in DL_MESO_DPD, as given in the `constants`, `variables` and `numeric_container` modules. Because DL_MESO is an ongoing project and new parameters might be added to the package in the future, it is strongly recommended that users of DL_MESO check the names of any self-defined variables whenever the package is updated to reduce the possibility of duplications causing unexpected errors.

The notation column in Table 9.4 gives the restrictions applicable on the parameters. 'A' indicates an array of data, followed by the number of elements in the array. For example, 'A `maxdim`' means the parameter is actually an array with `maxdim` elements (numbered from 1 to `maxdim`). '≥ 1' means the number must be greater or equal to one, while for a Boolean parameter 'T or F' means its value can either be `.true.` or `.false.`. An asterisk in the data type for the array indicates that it is allocatable, while two asterisks denote an allocatable array that is deallocated before DPD calculations start.

Table 9.4: DL_MESO_DPD Parameters

function	parameter	data type	notation
kind parameter for double precision numbers	dp	integer	
kind parameter for long integers	li	integer	
maximum number of particles	maxdim	integer	
maximum message buffer size	maxbuf	integer	
maximum data stack size	mxstak	integer	
maximum number of interaction parameters	mxprm	integer	
maximum number of species	mxspe	integer	
maximum number of interaction pairings	mxpot	integer	
maximum number of molecule definitions	mxmoldef	integer	
maximum particle size for molecule	mxmolsize	integer	
maximum number of bonds per molecule	mxbonds	integer	
restart file (export*) creation interval	ndump	integer	
length of double precision MPI message	Dlen	integer	
name of processing unit	idnode	integer	
number of processing units	nodes	integer	
filename for restart files	exportname	character(10)	
switch for temperature scaling	ltemp	logical	T or F
switch for reading CONFIG file	lconfig	logical	T or F
switch for writing CORREL file	lcorr	logical	T or F
switch for writing HISTORY* files	lsbsv	logical	T or F
switch for modelling bonds	lbond	logical	T or F
switch for modelling bond angles	langle	logical	T or F
switch for modelling bond dihedrals	ldihed	logical	T or F
switch for global holding of bond information	lgbnd	logical	T or F
switch for isotropic variations of volume with pressure	lisoprs	logical	T or F
printout selection for OUTPUT file	outsel	integer	
name of DL_MESO_DPD calculation	text	character(80)	
number of time steps for calculation	nrun	integer	
interval for writing OUTPUT file	nsbpo	integer	≥ 1
interval for writing CORREL file	iscorr	integer	
interval for writing HISTORY* files	nsbsv	integer	
temperature scaling interval	nsbts	integer	
number of equilibration time steps	nseql	integer	
calculation restart parameter	kres	integer	
number of species	nspe	integer	≥ 1, $\leq$mxspe
species population of unbonded particles	nspec	integer	A mxspe
number of defined molecule types	nmoldef	integer	$\leq$ mxmoldef
size of statistical data stack	nstk	integer	≥ 1, $\leq$mxstak
total number of particles in system	nsyst	integer	
total number of unbonded particles in system	nusyst	integer	
total number of molecules in system	nummol	integer	
total number of bonds in system	numbond	integer	
total number of bond angles in system	numang	integer	
total number of bond dihedrals in system	numdhd	integer	
number of defined bond types	nbonddef	integer	$\leq$ mxbonddef
number of defined bond angle types	nangdef	integer	$\leq$ mxbonddef
number of defined bond dihedral types	ndhddef	integer	$\leq$ mxbonddef
time step number	nstep	integer	
force calculation time accumulator	timfrc	real(KIND=dp)	
step time accumulator	timstp	real(KIND=dp)	
specified system temperature (k_BT)	temp	real(KIND=dp)	
size of time step (Δt)	tstep	real(KIND=dp)	
halo boundary size	rhalo	real(KIND=dp)	
interaction cutoff radius (r_c)	rcut	real(KIND=dp)	> 0
square of interaction cutoff radius	rct2	real(KIND=dp)	> 0
many-body DPD cutoff radius (r_d)	rmbcut	real(KIND=dp)	
square of many-body DPD cutoff radius	rmbct2	real(KIND=dp)	
short-range electrostatic cutoff (r_e)	relec	real(KIND=dp)	
square of electrostatic cutoff	rel2	real(KIND=dp)	
maximum calculation time	timjob	real(KIND=dp)	
calculation close time	tclose	real(KIND=dp)	
system volume	volm	real(KIND=dp)	
size of system in x-dimension	dimx	real(KIND=dp)	
size of system in y-dimension	dimy	real(KIND=dp)	
size of system in z-dimension	dimz	real(KIND=dp)	
number of domain cells in x	npx	integer	≥ 1
number of domain cells in y	npy	integer	≥ 1
number of domain cells in z	npz	integer	≥ 1
list of neighbouring processes	map	integer	A 6
x position of domain cell	idx	integer	
y position of domain cell	idy	integer	
z position of domain cell	idz	integer	
position of domain cell origin in system volume (x dimension)	delx	real(KIND=dp)	
position of domain cell origin in system volume (y dimension)	dely	real(KIND=dp)	
position of domain cell origin in system volume (z dimension)	delz	real(KIND=dp)	
domain cell length in x-direction	sidex	real(KIND=dp)	
domain cell length in y-direction	sidey	real(KIND=dp)	
domain cell length in z-direction	sidez	real(KIND=dp)	
number of particles in domain cell	nbeads	integer	
number of link cells in domain cell (x dimension)	nlx	integer	
number of link cells in domain cell (y dimension)	nly	integer	
number of link cells in domain cell (z dimension)	nlz	integer	
number of link cells in domain cell and boundary halo (x dimension)	nlx2	integer	

Table 9.4: DL_MESO_DPD Parameters (continued)

function	parameter	data type	notation
number of link cells in domain cell and boundary halo (y dimension)	nly2	integer	
number of link cells in domain cell and boundary halo (z dimension)	nlz2	integer	
link cell length in x-direction	wdthx	real(KIND=dp)	
link cell length in y-direction	wdthy	real(KIND=dp)	
link cell length in z-direction	wdthz	real(KIND=dp)	
number of link cells for electrostatics in domain cell (x dimension)	nlewx	integer	
number of link cells for electrostatics in domain cell (y dimension)	nlewy	integer	
number of link cells for electrostatics in domain cell (z dimension)	nlewz	integer	
number of link cells for electrostatics in domain cell and boundary halo (x dimension)	nlewx2	integer	
number of link cells for electrostatics in domain cell and boundary halo (y dimension)	nlewy2	integer	
number of link cells for electrostatics in domain cell and boundary halo (z dimension)	nlewz2	integer	
electrostatic link cell length in x-direction	wdthewx	real(KIND=dp)	
electrostatic link cell length in y-direction	wdthewy	real(KIND=dp)	
electrostatic link cell length in z-direction	wdthewz	real(KIND=dp)	
species name	namspe	character(8)	A mxspe
species particle mass	amass	real(KIND=dp)	A mxspe
potential interaction type	ktype	integer	
species interaction parameter (energy) A_i	aaa	real(KIND=dp)	A mxspe
species interaction parameter (distance) B_i	bbb	real(KIND=dp)	A mxspe
species interaction parameter (charge) C_i	ccc	real(KIND=dp)	A mxspe
species interaction parameter (dissipation) D_i (γ_i or Γ_i)	ddd	real(KIND=dp)	A mxspe
Lennard-Jones long-range potential correction	clr	real(KIND=dp)	A 2
interaction parameter storage	vvv	real(KIND=dp)	A mxprm,mxpot
unlike particle A_{ij} value	unlikeaaa	real(KIND=dp)**	A mxpot
unlike particle B_{ij} value	unlikebbb	real(KIND=dp)**	A mxpot
unlike particle γ_{ij} or Γ_{ij} value	unlikegam	real(KIND=dp)**	A mxpot
species number i for unlike particle A_{ij} value	unlikei	integer**	A mxpot
species number j for unlike particle A_{ij} value	unlikej	integer**	A mxpot
integrator/thermostat type	itype	integer	
thermostat parameter a	atherm	real(KIND=dp)	
thermostat parameter b	btherm	real(KIND=dp)	
thermostat parameter c	ctherm	real(KIND=dp)	
thermostat parameter d	dtherm	real(KIND=dp)	
dissipative coefficient (γ)/collision frequency (Γ)	gamma	real(KIND=dp)	A mxpot
random force parameter (σ)[1]	sigma	real(KIND=dp)	A mxpot
Lowe-Andersen probability of velocity rescaling ($\Gamma\Delta t$)	gammala	real (KIND=dp)	A mxpot
Peters dissipative coefficient (γ)	gammap	real (KIND=dp)	A mxpot
Stoyanov-Groot probability of velocity rescaling ($\Gamma\Delta t$)	gammasg	real (KIND=dp)	A mxpot
Stoyanov-Groot Nosé-Hoover coupling parameter (α)	alphasg	real(KIND=dp)	
instantaneous temperature nominator accumulator	itpnm	real(KIND=dp)	
instantaneous temperature denominator accumulator	itpdm	real(KIND=dp)	
thermostat pair list: particle i	pairlsti	integer*	A mxpair[2]
thermostat pair list: particle j	pairlstj	integer*	A mxpair
thermostat pair list: species of particle i	pairspei	integer*	A mxpair
thermostat pair list: species of particle j	pairspei	integer*	A mxpair
thermostat pair list: Maxwell distributed velocity (v_{ij}°)	veleq	real(KIND=dp)*	A mxpair
thermostat pair list: x component of interparticle distance	dxx	real(KIND=dp)*	A mxpair
thermostat pair list: y component of interparticle distance	dyy	real(KIND=dp)*	A mxpair
thermostat pair list: z component of interparticle distance	dzz	real(KIND=dp)*	A mxpair
number of particle pairs in thermostat pair list	npair	integer	
system-wide virial corrections	vrlsg	real(KIND=dp)*	A nsyst[3]
barostat type	btype	integer	
barostat target pressure (P_0)	prszero	real(KIND=dp)	
thermostat parameter a	atherm	real(KIND=dp)	
thermostat parameter b	btherm	real(KIND=dp)	
thermostat parameter c	ctherm	real(KIND=dp)	
thermostat parameter d	dtherm	real(KIND=dp)	
x component of piston velocity	upx	real(KIND=dp)	
y component of piston velocity	upy	real(KIND=dp)	
z component of piston velocity	upz	real(KIND=dp)	
x component of piston velocity at previous timestep	upx1	real(KIND=dp)	
y component of piston velocity at previous timestep	upy1	real(KIND=dp)	
z component of piston velocity at previous timestep	upz1	real(KIND=dp)	
piston mass (W_g)	psmass	real(KIND=dp)	
x component of piston force	fpx	real(KIND=dp)	
y component of piston force	fpy	real(KIND=dp)	
z component of piston force	fpz	real(KIND=dp)	
instantaneous virial	ivrl	real(KIND=dp)	A 3
Langevin random force parameter (σ_p)	sigmalang	real(KIND=dp)	
electrostatic algorithm type	etype	integer	
electrostatic interaction parameter a	aelec	real(KIND=dp)	
electrostatic interaction parameter b	belec	real(KIND=dp)	
electrostatic interaction parameter c	celec	real(KIND=dp)	
electrostatic interaction parameter d	delec	real(KIND=dp)	
electrostatic coupling parameter (Γ)	gammaelec	real(KIND=dp)	
total system charge ($\sum_i q_i$)	qchg	real(KIND=dp)	
Ewald real-space convergence parameter (α)	alphaew	real(KIND=dp)	

[1]This incorporates the time step for Velocity Verlet integration and is thus equal to $\sqrt{\frac{2\gamma k_B T}{\Delta t}}$.

[2]This non-global value is determined according to the number of particles and link cells required: not required for DPD thermostat (MD-VV or DPD-VV).

[3]Only required for Stoyanov-Groot thermostat.

Table 9.4: DL_MESO_DPD Parameters (continued)

function	parameter	data type	notation
reciprocal of real-space convergence parameter	ralphaew	real(KIND=dp)	
Slater charge smearing coefficient (β)	betaew	real(KIND=dp)	
maximum reciprocal vector size in x dimension (k_1^{max})	kmax1	integer	≥ 1[4]
maximum reciprocal vector size in y dimension (k_2^{max})	kmax2	integer	
maximum reciprocal vector size in z dimension (k_3^{max})	kmax3	integer	
Ewald self-interaction correction	engsic	real(KIND=dp)	
charged system correction	qfixv	real(KIND=dp)	
bond interaction parameter a	aabond	real(KIND=dp)	A mxbonddef
bond interaction parameter b	bbbond	real(KIND=dp)	A mxbonddef
bond interaction parameter c	ccbond	real(KIND=dp)	A mxbonddef
bond interaction parameter d	ddbond	real(KIND=dp)	A mxbonddef
angle interaction parameter a	aaang	real(KIND=dp)	A mxbonddef
angle interaction parameter b	bbang	real(KIND=dp)	A mxbonddef
angle interaction parameter c	ccang	real(KIND=dp)	A mxbonddef
angle interaction parameter d	ddang	real(KIND=dp)	A mxbonddef
dihedral interaction parameter a	aadhd	real(KIND=dp)	A mxbonddef
dihedral interaction parameter b	bbdhd	real(KIND=dp)	A mxbonddef
dihedral interaction parameter c	ccdhd	real(KIND=dp)	A mxbonddef
dihedral interaction parameter d	dddhd	real(KIND=dp)	A mxbonddef
bond types	bdtype	integer	A mxbonddef
bond angle types	angtype	integer	A mxbonddef
bond dihedral types	dhdtype	integer	A mxbonddef
bond table	bndtbl	integer*	A numbond,3
bond angle table	angtbl	integer*	A numang,4
bond dihedral table	dhdtbl	integer*	A numdhd,5
global/local particle number list	lblclst	integer*	A 4*(nsyst-nusyst),2
number of bonds in table	nbonds	integer	
number of bond angles in table	nangles	integer	
number of bond dihedrals in table	ndiheds	integer	
number of entries in global/local particle number list	nlist	integer	
species population of bonded particles	nspecmol	integer	A mxspe
molecule name	nammol	character(8)	A mxmoldef
molecule type population	nmol	integer	A mxmoldef
bead numbers in molecule types	nbdmol	integer	A mxmoldef
species number for molecule insertion	mlstrtspe	integer**	A nmoldef,mxmolsize
x coordinate for molecule insertion	mlstrtxxx	real(KIND=dp)**	A nmoldef,mxmolsize
y coordinate for molecule insertion	mlstrtyyy	real(KIND=dp)**	A nmoldef,mxmolsize
z coordinate for molecule insertion	mlstrtzzz	real(KIND=dp)**	A nmoldef,mxmolsize
cube size for molecule insertion	cbsize	real(KIND=dp)**	A nmoldef
number of bonds for molecule type	nbond	integer**	A nmoldef
number of bond angles for molecule type	nangle	integer**	A nmoldef
number of bond dihedrals for molecule type	ndihed	integer**	A nmoldef
bond table storage for molecule insertion	bdinp1	integer**	A nmoldef,mxbonds
bond table storage for molecule insertion	bdinp2	integer**	A nmoldef,mxbonds
bond table storage for molecule insertion	bdinp3	integer**	A nmoldef,mxbonds
angle table storage for molecule insertion	anginp1	integer**	A nmoldef,mxbonds
angle table storage for molecule insertion	anginp2	integer**	A nmoldef,mxbonds
angle table storage for molecule insertion	anginp3	integer**	A nmoldef,mxbonds
angle table storage for molecule insertion	anginp4	integer**	A nmoldef,mxbonds
dihedral table storage for molecule insertion	dhdinp1	integer**	A nmoldef,mxbonds
dihedral table storage for molecule insertion	dhdinp2	integer**	A nmoldef,mxbonds
dihedral table storage for molecule insertion	dhdinp3	integer**	A nmoldef,mxbonds
dihedral table storage for molecule insertion	dhdinp4	integer**	A nmoldef,mxbonds
dihedral table storage for molecule insertion	dhdinp5	integer**	A nmoldef,mxbonds
many-body DPD interaction parameter A_{ij}	aamb	real(KIND=dp)	A mxpot
many-body DPD interaction parameter B_{ij}	bbmb	real(KIND=dp)	A mxpot
many-body DPD interaction parameter C_{ij}	ccmb	real(KIND=dp)	A mxpot
many-body DPD interaction parameter D_{ij}	ddmb	real(KIND=dp)	A mxpot
many-body DPD interaction parameter E_{ij}	eemb	real(KIND=dp)	A mxpot
localized densities	rhomb	real(KIND=dp)	A maxdim, mxspe
x component of external body force per particle	bdfrcx	real(KIND=dp)	A mxspe
y component of external body force per particle	bdfrcy	real(KIND=dp)	A mxspe
z component of external body force per particle	bdfrcz	real(KIND=dp)	A mxspe
species name for particle	atmnam	character(8)	A maxdim
molecule name for particle	molnam	character(8)	A maxdim
particle global identity number	lab	integer	A maxdim
particle species number	ltp	integer	A maxdim
particle molecule type number	ltm	integer	A maxdim
particle link cell population number	lct	integer	A maxdim
particle link cell number	link	integer	A maxdim
particle local domain cell identity number	loc	integer	A maxdim
particle molecule type number	ltm	integer	A maxdim
force x-component on particle	fxx	real(KIND=dp)	A maxdim
force y-component on particle	fyy	real(KIND=dp)	A maxdim
force z-component on particle	fzz	real(KIND=dp)	A maxdim
variable force x-component on particle	fvx	real(KIND=dp)	A maxdim
variable force y-component on particle	fvy	real(KIND=dp)	A maxdim
variable force z-component on particle	fvz	real(KIND=dp)	A maxdim
velocity x-component of particle	vxx	real(KIND=dp)	A maxdim
velocity y-component of particle	vyy	real(KIND=dp)	A maxdim
velocity z-component of particle	vzz	real(KIND=dp)	A maxdim

[4]Requirement only applicable if Ewald sum electrostatics are used.

Table 9.4: DL_MESO_DPD Parameters (continued)

function	parameter	data type	notation
Cartesian coordinate x for particle	xxx	real(KIND=dp)	A maxdim
Cartesian coordinate y for particle	yyy	real(KIND=dp)	A maxdim
Cartesian coordinate z for particle	zzz	real(KIND=dp)	A maxdim
particle mass	weight	real(KIND=dp)	A maxdim
virial of particle	vrl	real(KIND=dp)	A maxdim
potential energy of particle	pot	real(KIND=dp)	A maxdim
potential energy accumulator	pe	real(KIND=dp)	
virial accumulator	vir	real(KIND=dp)	
kinetic energy accumulator	tke	real(KIND=dp)	
bond potential energy accumulator	be	real(KIND=dp)	
angle potential energy accumulator	ae	real(KIND=dp)	
dihedral potential energy accumulator	de	real(KIND=dp)	
electrostatic potential energy accumulator	ee	real(KIND=dp)	
bond length accumulator	bdlng	real(KIND=dp)	
bond angle accumulator	bdang	real(KIND=dp)	
bond dihedral accumulator	bddhd	real(KIND=dp)	
average system potential energy	avepe	real(KIND=dp)	
average system virial	avevir	real(KIND=dp)	
average system kinetic energy	avetke	real(KIND=dp)	
average system total energy	avete	real(KIND=dp)	
average system pressure	aveprs	real(KIND=dp)	
average system volume	avevlm	real(KIND=dp)	
average system temperature	avettp	real(KIND=dp)	
average system bond potential energy	avebe	real(KIND=dp)	
average system angle potential energy	aveae	real(KIND=dp)	
average system dihedral potential energy	avede	real(KIND=dp)	
average system electrostatic potential energy	aveee	real(KIND=dp)	
system potential energy fluctuation	flcpe	real(KIND=dp)	
system virial fluctuation	flcvir	real(KIND=dp)	
system kinetic energy fluctuation	flcke	real(KIND=dp)	
system total energy fluctuation	flcte	real(KIND=dp)	
system pressure fluctuation	flcprs	real(KIND=dp)	
system volume fluctuation	flcvlm	real(KIND=dp)	
system temperature fluctuation	flcttp	real(KIND=dp)	
system bond potential energy fluctuation	flcbe	real(KIND=dp)	
system angle potential energy fluctuation	flcae	real(KIND=dp)	
system dihedral potential energy fluctuation	flcde	real(KIND=dp)	
system electrostatic potential energy fluctuation	flcee	real(KIND=dp)	
system potential energy accumulator	zumpe	real(KIND=dp)	
system virial accumulator	zumvir	real(KIND=dp)	
system kinetic energy accumulator	zumtke	real(KIND=dp)	
system volume accumulator	zumvlm	real(KIND=dp)	
system bond potential energy accumulator	zumbe	real(KIND=dp)	
system angle potential energy accumulator	zumae	real(KIND=dp)	
system dihedral potential energy accumulator	zumde	real(KIND=dp)	
system electrostatic potential energy accumulator	zumee	real(KIND=dp)	
system potential energy at current step	stppe	real(KIND=dp)	
system virial at current step	stpvir	real(KIND=dp)	
system kinetic energy at current step	stptke	real(KIND=dp)	
system total energy at current step	stptke	real(KIND=dp)	
system pressure at current step	stpprs	real(KIND=dp)	
system volume at current step	stpvlm	real(KIND=dp)	
system temperature at current step	stpttp	real(KIND=dp)	
system bond potential energy at current step	stpbe	real(KIND=dp)	
system angle potential energy at current step	stpae	real(KIND=dp)	
system dihedral energy at current step	stpde	real(KIND=dp)	
system electrostatic energy at current step	stpee	real(KIND=dp)	
system mean bond length at current step	stpbdl	real(KIND=dp)	
system mean bond angle at current step	stpang	real(KIND=dp)	
system mean bond dihedral at current step	stpdhd	real(KIND=dp)	
rolling average system potential energy	ravpe	real(KIND=dp)	
rolling average system virial	ravvir	real(KIND=dp)	
rolling average system kinetic energy	ravtke	real(KIND=dp)	
rolling average system total energy	ravte	real(KIND=dp)	
rolling average system pressure	ravprs	real(KIND=dp)	
rolling average system volume	ravvlm	real(KIND=dp)	
rolling average system temperature	ravttp	real(KIND=dp)	
rolling average system bond potential energy	ravbe	real(KIND=dp)	
rolling average system angle potential energy	ravae	real(KIND=dp)	
rolling average system dihedral potential energy	ravde	real(KIND=dp)	
rolling average system electrostatic potential energy	ravee	real(KIND=dp)	
stage number for data stack	nav	integer	
data stack for potential energy	stkpe	real(KIND=dp)	A mxstak
data stack for virial	stkvir	real(KIND=dp)	A mxstak
data stack for kinetic energy	stktke	real(KIND=dp)	A mxstak
data stack for volume	stkvlm	real(KIND=dp)	A mxstak
data stack for bond potential energy	stkbe	real(KIND=dp)	A mxstak
data stack for angle potential energy	stkae	real(KIND=dp)	A mxstak
data stack for dihedral potential energy	stkde	real(KIND=dp)	A mxstak
data stack for electrostatic potential energy	stkee	real(KIND=dp)	A mxstak
duni random number generator state	uni	integer	A 102
mtrnd random number generator state	mt	integer	A 0:624

Chapter 10

DL_MESO_DPD Features

10.1 Domain decomposition and linked-list cell calculations

The Domain Decomposition (DD) strategy is one of several ways to parallelize particle-based simulations[45]. Its basis is the division of the simulated system into equal-sized spatial blocks or domains, each of which is allocated to a specific processing unit of a parallel computer. The arrays defining the coordinates, velocities and forces for all N particles in the system are divided into sub-arrays of size $\approx \frac{N}{P}$ on each of the P processing units, with the particles allocated geometrically among them. In order for the strategy to work efficiently, the simulated system should possess a reasonably uniform density so that each processing unit is allocated as equal a portion of particle data as possible. The computation of forces and integration of the equations of motion are shared (more or less) equally between the processing units and to a large extent can be computed independently on each unit. While tricky to program, this method is conceptually simple and particularly suited to large-scale simulations.

The DD strategy which underpins DL_MESO_DPD is based on the link cell algorithm[19], which requires a relatively short-ranged cutoff for interparticle potentials and forces. There is a need for processing units to exchange 'halo data', i.e. sending the contents of link cells at the boundaries of each domain to neighbouring units so each may have all the necessary information to compute pairwise forces acting on the particles in its allotted domain. Similarly the force and virial contributions from particles in boundary halos need to be returned to their original processing units for summation. The link cell algorithm is also applied in serial by duplicating system data to create the boundary halo across periodic boundaries.

The size of the boundary halo (`rhalo`) – which can be specified in the `CONTROL` file – should not be greater than the minimum system dimension per domain; for good parallel performance, it is recommended that the halo size should be no larger than one-third of the smallest subdomain dimension. The value of `maxdim` in the `constants` module giving the maximum sizes of force, velocity and position arrays should be large enough to hold all particles in each domain *plus* any particles in boundary halos, including duplicates when running in serial or using smaller numbers of processing units.

10.1.1 Intramolecular interactions

Intramolecular interactions may be handled in two different ways: either (1) locally with each processing unit being allocated a subset of bonds to deal with (including bonds across neighbouring units), or (2) globally with all units holding all bond data and sharing bonded particle positions, each carrying out all bond calculations and appropriately allocating forces to local particles. The former method may require larger boundary halo sizes for the bond lengths being simulated but is more efficient for larger numbers of molecules and processing units, while the latter method requires the sharing of information between all units but does not require halo information and is guaranteed to find all bonds.

Bookkeeping arrays (`bndtbl`, `angtbl` and `dhdtbl`) list all particles involved in bonded interactions according to global index numbers and point to appropriate arrays of parameters to define the potential. If the 'key' bonded particle for a bond[1] moves from one processing unit to another, the entry in the bookkeeping array is also moved. At each time step a list of bonded particles in each domain (`lblclst`) is created to relate global index numbers to the local index numbers used by the processing unit in force, velocity and coordinate arrays. This global/local index list is sorted by global index number to allow cross-referencing to local index numbers by means of a binary search.

10.1.2 Electrostatic interactions

For systems with periodic boundary conditions DL_MESO_DPD uses the Ewald Sum to calculate Coulombic interactions (see Section 10.5). Calculation of the real space component (in routine `ewald_real`) uses the same link cell algorithm as for other pairwise interactions, albeit using a larger cutoff radius (r_e) and requires a larger boundary halo than for standard pairwise interactions.

10.2 Thermostats and integration algorithms

The integration algorithms in DL_MESO_DPD are based on the second-order Velocity Verlet (VV) scheme[54], which yields the positions, velocities and forces of particles at the same time and is generally used in molecular dynamics simulations. This algorithm has two stages. The first stage advances the particle velocities to time $t + \frac{1}{2}\Delta t$ by integrating the forces and uses the new half-step velocities to advance the position to time $t + \Delta t$:

$$\vec{v}_i\left(t + \tfrac{1}{2}\Delta t\right) = \vec{v}_i\left(t\right) + \frac{\Delta t}{2}\frac{\vec{F}_i\left(t\right)}{m_i} \tag{10.1}$$

$$\vec{r}_i\left(t + \Delta t\right) = \vec{r}_i\left(t\right) + \Delta t\vec{v}_i\left(t + \tfrac{1}{2}\Delta t\right) \tag{10.2}$$

The positions at the end of the time step allow the forces to be recalculated, before the second stage of the algorithm is applied to advance the half-step velocities to the end of the time step by integrating with the new force:

$$\vec{v}_i\left(t + \Delta t\right) = \vec{v}_i\left(t + \tfrac{1}{2}\Delta t\right) + \frac{\Delta t}{2}\frac{\vec{F}_i\left(t + \Delta t\right)}{m_i} \tag{10.3}$$

Five thermostating algorithms are currently available in DL_MESO_DPD: two variants of the standard DPD thermostat and three alternative schemes which apply velocity corrections to the particles after force integration. The algorithm can be selected in the `CONTROL` file using the keyword `THERM`: `itype` is an integer specifying the thermostat/integration algorithm and a, b, c and d are system-wide parameters whose meanings for each thermostat are given in Table 10.1.

Table 10.1: Thermostat coefficients used in DL_MESO_DPD

`itype`	a^2	b	c	d
0 (MD-VV)	γ	-	-	-
1 (DPD-VV)	γ	-	-	-
2 (Lowe-Andersen)	Γ	-	-	-
3 (Peters)	γ	-	-	-
4 (Stoyanov-Groot)	Γ	α	-	-

10.2.1 DPD thermostat with standard Velocity Verlet integration (MD-VV) (`itype=0`)

This algorithm uses the drag (dissipative) and random forces, $\vec{F}_{ij}^{D}$ and $\vec{F}_{ij}^{R}$ respectively as described in Chapter 8, as the system thermostat. These forces are combined with all other forces between particles – pairwise

[1] This is the first referenced particle in stretching bonds and the second for bond angles and dihedrals.

[2] These values can also be specified using the keyword `GAMMA` in the `CONTROL` file: the maximum of the two values is used.

conservative (standard and/or density-dependent), bonding, electrostatic, planar surface, external (body) forces – and integrated using the standard Velocity Verlet integrator.

The combination of the DPD thermostat with the standard MD-type VV algorithm is the simplest and least time-consuming thermostating algorithm available in DL_MESO_DPD. (If no integrator/thermostat type is selected in the `CONTROL` file, DL_MESO_DPD will use this algorithm by default.) The drag force does, however, depend upon particle velocities and is therefore only approximated using the mid-step values: this frequently produces a system temperature higher than that specified by the user and requires a small time step Δt to reduce the offset to tolerable levels.

10.2.2 DPD thermostat with DPD Velocity Verlet integration (DPD-VV) (`itype=1`)

As with the MD-VV scheme, this algorithm uses the drag and random forces as the system thermostat, which are combined with all other forces before being integrated using the Velocity Verlet scheme. The drag force is subsequently recalculated after the second stage using the velocities at the end of the time step[10].

The recalculation of drag forces after force integration helps to alleviate the temperature offset produced by the MD-VV, and hence larger time steps may be used for reasonable temperature control. It does require the re-use of the linked-list cells and inter-processor communications to recalculate the drag forces, which can significantly increase the time required per time step compared to the MD-VV scheme.

10.2.3 Lowe-Andersen thermostat (`itype=2`)

The Lowe-Andersen thermostat[27] is an alternative to the use of drag and random forces in the DPD thermostat, which uses a variant of the Andersen thermostat[1]. After all other forces (conservative, bonding etc.) are integrated using the Velocity Verlet scheme, a random sample of particle pairs have their relative velocity replaced by a value from a Maxwellian distribution, i.e.

$$v_{ij}^{\circ} = \zeta_{ij}\sqrt{\frac{k_B T}{\mu_{ij}}} \tag{10.4}$$

where $\mu_{ij} = \frac{m_i m_j}{m_i + m_j}$ is the reduced mass between the two particles. The velocities of particles i and j thus become:

$$\vec{v}_i = \vec{v}_i + \frac{\mu_{ij}}{m_i}\left(-\left(\hat{e}_{ij}\cdot\vec{v}_{ij}\right) + v_{ij}^{\circ}\right)\hat{e}_{ij} \tag{10.5}$$

$$\vec{v}_j = \vec{v}_j - \frac{\mu_{ij}}{m_i}\left(-\left(\hat{e}_{ij}\cdot\vec{v}_{ij}\right) + v_{ij}^{\circ}\right)\hat{e}_{ij} \tag{10.6}$$

The probability of a particle pair being thermostatted is equal to $\Gamma\Delta t$, where Γ is defined as the collision frequency (with a maximum value of $\frac{1}{\Delta t}$), and the velocity corrections to particle pairs are applied in a random order to prevent biasing.

The viscosity and self-diffusion generated by this thermostat for a single species are

$$\nu = \frac{\pi\rho\Gamma r_c^5}{75m} \tag{10.7}$$

$$D = \frac{k_B T \tau_D}{m} \tag{10.8}$$

where τ_D is the decay time for velocity correlations and inversely proportional to the collision frequency. The Schmidt number is therefore proportional to $\frac{\Gamma^2}{k_B T}$ and can thus reach values up to $\mathcal{O}(10^5)$.

This thermostat is suited to systems with higher viscosities and low diffusitivies while giving the correct system temperature for a wide range of time step sizes (within numerical errors due to Velocity Verlet force integration). Its implementation in parallel running uses a replicated data strategy to carry out the velocity corrections: this requires additional memory on each processing unit for the velocities of all particles and the data required to modify the velocities of particle pairs. The efficiency of the Lowe-Andersen thermostat thus decreases with increasing numbers of particles in the entire system and Γ.

10.2.4 Peters thermostat (itype=3)

The Peters thermostat[38] is a modification of the Lowe-Andersen thermostat that reduces to standard DPD as the time step tends to zero. After integrating all forces using the Velocity Verlet scheme, all particle pairs have their velocities modified (in a random order) using:

$$\vec{v}_i = \vec{v}_i + \frac{1}{m_i}\left(-a_{ij}\left(\hat{e}_{ij}\cdot\vec{v}_{ij}\right)\Delta t + b_{ij}\zeta_{ij}\sqrt{\Delta t}\right)\hat{e}_{ij} \tag{10.9}$$

$$\vec{v}_j = \vec{v}_j - \frac{1}{m_i}\left(-a_{ij}\left(\hat{e}_{ij}\cdot\vec{v}_{ij}\right)\Delta t + b_{ij}\zeta_{ij}\sqrt{\Delta t}\right)\hat{e}_{ij} \tag{10.10}$$

where the coefficients a_{ij} and b_{ij} are chosen so that

$$b_{ij} = \sqrt{2k_BTa_{ij}\left(1-\frac{a_{ij}\Delta t}{2\mu_{ij}}\right)}.$$

To ensure that the thermostat both reduces to the DPD thermostat as the time step reduces to zero and is not restricted by the choice of time step, the coefficients are chosen as follows:

$$a_{ij} = \frac{\mu_{ij}}{\Delta t}\left(1-\exp\left[-\frac{\gamma_{ij}\omega(r_{ij})\Delta t}{\mu_{ij}}\right]\right) \tag{10.11}$$

$$b_{ij} = \sqrt{\frac{k_BT\mu_{ij}}{\Delta t}\left(1-\exp\left[-\frac{2\gamma_{ij}\omega(r_{ij})\Delta t}{\mu_{ij}}\right]\right)} \tag{10.12}$$

A correction to the virial of $\left(a_{ij}\left(\hat{e}_{ij}\cdot\vec{v}_{ij}\right) - b_{ij}\zeta_{ij}\sqrt{\Delta t}\right)r_{ij}^2$ is also applied for each particle pair.

This thermostat can be used with larger time steps than the standard DPD thermostat but with similarly low system viscosities. As for the Lowe-Andersen thermostat, its implementation in parallel running uses a replicated data strategy to carry out the velocity corrections, which requires additional memory per processing unit for storing the velocities of all particles in the system and the data required to modify them. The efficiency of the Peters thermostat therefore depends upon the total number of particles in the system: since all particle pairs are modified, calculation times for this thermostat may be comparable to those for the Lowe-Andersen thermostat when $\Gamma\Delta t \approx 1$.

10.2.5 Stoyanov-Groot thermostat (itype=4)

The Stoyanov-Groot thermostat[49] is a combination of the Lowe-Andersen thermostat and a Galilean-invariant Nosé-Hoover thermostat which acts locally and on pairs of particles. During force calculations after the first Velocity Verlet stage, the choice to use either the Lowe-Andersen or Nosé-Hoover thermostats for each particle pair is made at random; the Lowe-Andersen thermostat is selected with a probability of $\Gamma\Delta t$. The system temperature is also determined in terms of relative velocities for all particle pairs, i.e.

$$k_BT^* = \frac{\sum_{i>j}\psi^T(r_{ij})\mu_{ij}\vec{v}_ij^2}{3\sum_{i>j}\psi^T(r_{ij})} \tag{10.13}$$

where $\psi^T(r_{ij})$ is a smearing function for the temperature, chosen to reduce to zero when $r_{ij} > r_c$: by default this is set as $\psi^T(r_{ij}) = 1$ for $r < r_c$. For all particle pairs that are to be subjected to the Nosé-Hoover thermostat, an additional thermostating force is included:

$$\vec{F}_{ij}^T = -\alpha w^T(r_{ij})\left(1-\frac{k_BT^*}{k_BT}\right)\left[\vec{v}_{ij}\cdot\hat{e}_{ij}\right]\hat{e}_{ij} \tag{10.14}$$

with α as a system-wide thermostat coupling parameter and $w^T(r_{ij})$ as a switching function, which by default is equivalent to $w^R(r_{ij}) = 1 - \frac{r_{ij}}{r_c}$ for standard DPD. (A virial correction of $-\vec{F}_{ij}^T\cdot\vec{r}_{ij}$ is also made for each particle pair.) All other particle pairs are thermostated using the standard Lowe-Andersen method.

This thermostat can produce a wide range of system viscosities and diffusivities with good temperature control and hydrodynamics, using the collision frequency Γ to obtain the required Schmidt number. The replicated

data strategy is again used for the Lowe-Andersen part, which requires memory in each processing unit to store the velocities of all particles and the data for particle pair modification using the Lowe-Andersen scheme: the Nosé-Hoover scheme calculates the thermostating forces locally and does not require additional memory.

10.3 Barostats

In addition to a thermostat, a barostat may be included in simulations to obtain a desired average pressure (P_0) by adjusting the size (and shape) of the simulation cell. DL_MESO_DPD includes two such algorithms: a Langevin-type barostat[22] and the Berendsen barostat[2], both of which have been coupled to all five available thermostats.

The instantaneous isotropic pressure in a system is calculated using the virial theorem:

$$P(t) = \frac{1}{3V(t)} \left[\sum_i m_i v_i^2(t) + \sum_i \vec{F}_i(t) \cdot \vec{r}_i(t) \right] \tag{10.15}$$

while for anisotropic orthorhombic systems the pressure in dimension α, related to the instantaneous stress tensor component $\sigma_{\alpha\alpha}(t)$, is defined as

$$P_\alpha(t) = \frac{1}{V(t)} \left[\sum_i m_i v_{i,\alpha}^2(t) + \sum_i F_{i,\alpha}(t) r_{i,\alpha}(t) \right]. \tag{10.16}$$

All barostat definitions are expressed for the more general anisotropic case: these can be applied for isotropic systems by setting $P_x(t)$, $P_y(t)$ and $P_z(t)$ all equal to $P(t)$. The flag `lisoprs` can be used to determine whether the system pressure is isotropic or not.

The barostat can be selected in the `CONTROL` file using the keyword `BARO`: `btype` is an integer specifying the thermostat/integration algorithm and a, b, c and d are system-wide parameters whose meanings for each thermostat are given in Table 10.2.

Table 10.2: Barostat coefficients used in DL_MESO_DPD

`btype`	a	b	c	d
1 (Langevin)	τ_p	γ_p	-	-
2 (Berendsen)	$\frac{\beta}{\tau_p}$	-	-	-

10.3.1 Langevin barostat (`btype=1`)

The governing equation for the Langevin barostat on an orthorhombic simulation cell[22] is the force exerted by the piston (expressed as the time-derivative of its momentum $p_{g,\alpha} = W_g u_{g,\alpha}$):

$$\dot{p}_{g,\alpha} = V(P_\alpha - P_0) + \frac{1}{N_f} \sum_i m_i v_i^2 - \gamma_p p_{g,\alpha} + \sigma_p \zeta_{p,\alpha} \tag{10.17}$$

where N_f is the number of degrees of freedom: for a three-dimensional box containing N moving particles, $N_f = 3(N-1)$. γ_p and $\sigma_p \equiv \sqrt{2\gamma_p W_g k_B T}$ are respectively the drag and random coefficients for the piston and $\zeta_{p,\alpha}$ is a Gaussian random number for dimension α (this is set to the same value for all three dimensions if operating isotropically). When both γ_p and σ_p are set to zero, the Langevin barostat reduces to the **extended system** method.

The subsequent simulation cell size L_α can be determined by

$$\dot{L}_\alpha = \frac{p_{g,\alpha} L_\alpha}{W_g} = u_{g,\alpha} L_\alpha \tag{10.18}$$

with the barostat mass W_g chosen to be equal to $\frac{1}{3} N k_B T \tau_p^2$, where τ_p is the characteristic barostat time and should be set equal to between $\frac{2}{\gamma_p}$ and $\frac{10}{\gamma_p}$.

The velocities and positions of the particles are calculated by integration of slightly modified differential equations:

$$\frac{dv_{i,\alpha}}{dt} = \frac{F_{i,\alpha}}{m_i} - u_{g,\alpha}v_{i,\alpha} - \frac{1}{N_f}v_{i,\alpha}\sum_{\alpha}u_{g,\alpha} \tag{10.19}$$

$$\frac{dr_{i,\alpha}}{dt} = v_{i,\alpha} + u_{g,\alpha}r_{i,\alpha} \tag{10.20}$$

where the force on particle i, $\vec{F}_i$, includes any thermostating forces: the time integral of these forces can be determined for all thermostat types.

The implementation of this barostat is carried out using the Velocity Verlet scheme to integrate the equations of motion for both the particles and the barostat. The first Velocity Verlet stage integrates the forces on the particles

$$v_{i,\alpha}\left(t+\tfrac{1}{2}\Delta t\right) = v_{i,\alpha}\left(t\right) + \frac{\Delta t}{2}\frac{F_{i,\alpha}\left(t\right)}{m_i} - \Delta t u_{g,\alpha}v_{i,\alpha} \tag{10.21}$$

which is followed by a similar integration for the barostat velocity:

$$u_{g,\alpha}\left(t+\tfrac{1}{2}\Delta t\right) = u_{g,\alpha}\left(t\right) + \frac{\Delta t}{2}\frac{F_{g,\alpha}\left(t\right)}{W_g} \tag{10.22}$$

before the positions and simulation box dimensions are updated:

$$r_{i,\alpha}\left(t+\Delta t\right) = \exp\left(u_{g,\alpha}\left(t+\tfrac{1}{2}\Delta t\right)\Delta t\right)\left\{r_{i,\alpha}\left(t\right) + \Delta t v_{i,\alpha}\left(t+\tfrac{1}{2}\Delta t\right)\right\} \tag{10.23}$$

$$L_{\alpha}\left(t+\Delta t\right) = \exp\left(u_{g,\alpha}\left(t+\tfrac{1}{2}\Delta t\right)\Delta t\right)L_{\alpha}\left(t\right) \tag{10.24}$$

At this point the forces at the end of the time step are calculated (including thermostating forces), along with the system pressure. Since the the barostat force requires correct velocities for both the particles and barostat, an iterative procedure is required which begins by calculating an initial guess for the barostat velocity at the end of the time step:

$$u_{g,\alpha}^{(0)}\left(t+\Delta t\right) = u_{g,\alpha}\left(t-\Delta t\right) + \frac{2F_{g,\alpha}\left(t\right)\Delta t}{W_g}$$

Each iteration starts by calculating the particle velocities in a slightly modified second Velocity Verlet step:

$$v_{i,\alpha}^{(n+1)}\left(t+\Delta t\right) = \frac{\exp\left(u_{g,\alpha}\left(t+\frac{1}{2}\Delta t\right)\Delta t\right)v_{i,\alpha}\left(t+\frac{1}{2}\Delta t\right) + \frac{\Delta t}{2}\frac{F_{i,\alpha}(t+\Delta t)}{m_i}}{1+u_{g,\alpha}^{(n)}\Delta t} \tag{10.25}$$

The barostat force is then calculated using the same Gaussian random numbers for each iteration:

$$F_{g,\alpha}^{(n+1)}\left(t+\Delta t\right) = V(P_{\alpha}-P_0) + \frac{1}{N_f}\sum_i m_i\left(v_i^{(n+1)}\right)^2 - \gamma_p u_{g,\alpha}^{(n)}W_g + \sigma_p\zeta_{p,\alpha} \tag{10.26}$$

and the barostat velocity is recalculated:

$$u_{g,\alpha}^{(n+1)}\left(t+\Delta t\right) = u_{g,\alpha}\left(t+\tfrac{1}{2}\Delta t\right) + \frac{F_{g,\alpha}^{(n+1)}\left(t+\Delta t\right)\Delta t}{2W_g} \tag{10.27}$$

Equations 10.25 to 10.27 are repeated until convergence in particle velocities is achieved. This normally takes a few iterations per time step without requiring recalculation of particle forces or rescaling of particle coordinates.

10.3.2 Berendsen barostat (`btype=2`)

The governing equation for the Berendsen barostat[2] is a simple differential equation for the pressure:

$$\frac{dP_{\alpha}}{dt} = \frac{P_0 - P_{\alpha}}{\tau_p} \tag{10.28}$$

which can be solved to give a scaling factor for the simulation volume, $\vec{\eta}(t)$:

$$\eta_{\alpha}(t) = 1 - \frac{\beta\Delta t}{\tau_p}\left(P_0 - P_{\alpha}(t)\right) \tag{10.29}$$

where β is the isothermal compressibility of the system. The exact value of this property is not critical to the algorithm, since it relies on the ratio $\frac{\beta}{\tau_p}$.

The barostat is implemented using a variant of the Velocity Verlet algorithm; after the midstep velocities are determined, the scaling factor for time t is used to modify the particle positions and resize the simulation volume

$$r_{i,\alpha}\left(t+\Delta t\right) = \eta_\alpha(t) r_{i,\alpha}\left(t\right) + \Delta t v_{i,\alpha}\left(t+\tfrac{1}{2}\Delta t\right) \quad (10.30)$$

$$L_\alpha\left(t+\Delta t\right) = \eta_\alpha(t) L_\alpha\left(t\right) \quad (10.31)$$

The remainder of the Velocity Verlet algorithm is unchanged, although the scaling factor for the beginning of the next time step can be calculated at this point using Equation 10.29. No iteration is required for this barostat.

10.4 Particle-particle interactions

Pairwise particle interaction parameters in DL_MESO_DPD are specified in the `CONTROL` file for each species (A_α) and are stored in the array `aaa(i)`. Other properties that are also required for each species are its name (up to 8 characters) stored in `namspe(i)`, the number of unbonded particles of that species `nspec(i)`, the particle mass `amass(i)`, radius (B_i) `bbb(i)` and charge (q_i) `ccc(i)`.

The interaction parameter values and lengthscales are stored in the array `vvv(k, 1:4)` in preparation for DPD calculations, taking advantage of the fact that $A_{\alpha\beta} = A_{\beta\alpha}$ by converting the species numbers i and j into a potential number, $k = \frac{j(j-1)}{2} + i$. By default the conservative interaction parameter between different particle species α and β is set to the geometric mean, i.e.

$$A_{\alpha\beta} = \sqrt{A_\alpha A_\beta}$$

This choice can be overridden by specifying a replacement value for $A_{\alpha\beta}$ in the `CONTROL` file using the keyword `UNLIKE`.

By default, the interaction lengthscales between species α and β is set to the arithmetic mean, i.e. $B_{\alpha\beta} = \frac{1}{2}\left(B_\alpha + B_\beta\right)$. Like the interaction parameters, the default lengthscale can be overridden by specifying a replacement value for $B_{\alpha\beta}$ in the `CONTROL` file. It should be noted that the interaction length does *not* have to equal the interaction cut-off radius r_c (which applies for dissipative and random force interactions).

For all thermostats, the drag coefficient γ or collision frequency Γ is by default considered to be constant for all species and interactions between species. Individual values for the drag coefficient or collision frequency between the same and different species, $\gamma_{\alpha\beta}$ or $\Gamma_{\alpha\beta}$, can be specified in the `CONTROL` file to override this default.

Four types of pairwise interactions between particles are available in DL_MESO_DPD: Lennard-Jones, Weeks-Chandler-Andersen, Groot-Warren (standard) DPD and many-body (density-dependent) DPD. The keyword `KTYPE` in the `CONTROL` file determines which interactions are used.

10.4.1 Lennard-Jones (`ktype=0`)

The Lennard-Jones potential[23] is a mathematically simple model that approximates interactions (both attractive and repulsive) between pairs of neutral atoms or molecules:

$$U(r_{ij}) = 4\epsilon_{ij}\left[\left(\frac{\sigma_{ij}}{r_{ij}}\right)^{12} - \left(\frac{\sigma_{ij}}{r_{ij}}\right)^{6}\right] \quad (10.32)$$

where ϵ_{ij} is the depth of the potential well and σ_{ij} is the finite distance at which the potential is zero between particles i and j: in DL_MESO_DPD these are represented by A_{ij} and B_{ij} respectively. This potential and its related force are calculated for all interparticle distances (r_{ij}) less than the interaction cutoff radius r_c.

Long-range system-wide corrections to the potential and virial are required:

$$U^{lr} = \frac{4\pi}{V}\sum_{\alpha}\sum_{\beta\geq\alpha}(2-\delta_{\alpha\beta})N_\alpha N_\beta \epsilon_{\alpha\beta}\left(\frac{\sigma_{\alpha\beta}^{12}}{9r_c^9}-\frac{\sigma_{\alpha\beta}^{6}}{3r_c^3}\right) \tag{10.33}$$

$$\mathcal{W}^{lr} = -\frac{4\pi}{V}\sum_{\alpha}\sum_{\beta\geq\alpha}(2-\delta_{\alpha\beta})N_\alpha N_\beta \epsilon_{\alpha\beta}\left(\frac{12\sigma_{\alpha\beta}^{12}}{9r_c^9}-\frac{2\sigma_{\alpha\beta}^{6}}{r_c^3}\right) \tag{10.34}$$

where $\delta_{\alpha\beta}$ is the Kronecker delta (1 when $\alpha=\beta$, 0 when $\alpha\neq\beta$) and N_α the total number of particles of species α. These corrections multiplied by the volume are stored in the array `clr` to eliminate the need to adjust them if the system volume is changed by a barostat.

10.4.2 Weeks-Chandler-Andersen (`ktype=1`)

The Weeks-Chandler-Andersen potential[57] is a modification of the Lennard-Jones potential to produce purely repulsive, short-range interactions:

$$U(r_{ij}) = 4\epsilon_{ij}\left[\left(\frac{\sigma_{ij}}{r_{ij}}\right)^{12}-\left(\frac{\sigma_{ij}}{r_{ij}}\right)^{6}\right]+\epsilon_{ij}. \tag{10.35}$$

This interaction is applied for interparticle distances up to $2^{\frac{1}{6}}\sigma_{ij}$, which should be less than or equal to the interaction cutoff radius r_c. No long-range corrections to potential energy or virials are required.

10.4.3 Standard DPD (`ktype=2`)

The Groot-Warren (standard) form of DPD[13] uses the following purely repulsive, soft potential:

$$U\left(r_{ij}\right) = \frac{1}{2}A_{ij}\left(B_{ij}-r_{ij}\right)^2. \tag{10.36}$$

This conservative interaction is applied for interparticle distances up to B_{ij}, which should be less than or equal to r_c.

10.4.4 Many-body DPD (`ktype=3`)

The conservative force in standard DPD depends only upon the species interacting and the interparticle separation, which yields a quadratic equation of state. Many-body DPD[37, 53] is a method of providing alternative thermodynamic behaviours to DPD particles by making conservative forces additionally dependent on local densities.

The free energy of an inhomogeneous system with density $\rho(r)$ can be defined as the following in both continuous and ensemble-averaged discrete forms:

$$\mathcal{F} = \int d\vec{r}\rho(\vec{r})\psi(\overline{\rho}(\vec{r})) \tag{10.37}$$

$$= \left\langle\sum_i \psi(\overline{\rho}(\vec{r_i}))\right\rangle \tag{10.38}$$

where $\psi(\rho)$ is the free energy per particle in a homogeneous system and $\overline{\rho}(\vec{r_i})$ is a function related to the density at (and near) the position of particle i ($\vec{r_i}$). The latter can be approximated by a function dependent on the positions of particles close to particle i ($\tilde{\rho}_i$) to allow the calculation of an instantaneous free-energy:

$$\tilde{\mathcal{F}} = \sum_i \psi(\tilde{\rho}_i).$$

The effective (conservative) force on particle i can be obtained from the spatial derivative of the free energy, although only the excess part (equivalent to the potential energy U) is required since the kinetic motion of particles automatically accounts for the ideal contribution:

$$\vec{F}_i^C = -\frac{\partial \tilde{\mathcal{F}}^{ex}(\{\vec{r}_k\})}{\partial \vec{r}_i} = -\sum_j \frac{\partial \psi^{ex}(\tilde{\rho}_j)}{\partial \vec{r}_i} \tag{10.39}$$

The force can also be expressed in terms of pairwise interactions, taking a form similar to the standard DPD conservative force (Equation 8.5):

$$\vec{F}_{ij}^C = \left(\frac{\partial \psi^{ex}(\tilde{\rho}_i)}{\partial \tilde{\rho}_i} + \frac{\partial \psi^{ex}(\tilde{\rho}_j)}{\partial \tilde{\rho}_j} \right) w^C(r_{ij}) \frac{\vec{r}_{ij}}{r_{ij}} \tag{10.40}$$

The local-density approximation can be defined as a weighted average of instantaneous densities:

$$\begin{aligned} \tilde{\rho}_i &= \int d\vec{r}\, w^\rho(\mid \vec{r} - \vec{r}_i \mid) \rho(\vec{r}, \{\vec{r}_k\}) \\ &= \sum_{j \neq i} \int d\vec{r}\, w^\rho(\mid \vec{r} - \vec{r}_i \mid) \delta(\vec{r} - \vec{r}_j) \\ \tilde{\rho}_i &= \sum_{j \neq i} w^\rho(r_{ij}) \end{aligned} \tag{10.41}$$

with $w^\rho(r)$ as the weight function vanishing beyond a cutoff r_d (which can be equal to r_c or smaller) and normalized so that $\int_0^\infty 4\pi r^2 w^\rho(r) dr = 1$. The most frequently used form for the weight function is

$$w^\rho(r_{ij}) = \frac{15}{2\pi r_d^3} \left(1 - \frac{r_{ij}}{r_d} \right)^2 \qquad (r_{ij} < r_d) \tag{10.42}$$

which reduces to standard DPD when the excess free energy per particle is set to $\psi^{ex}(\tilde{\rho}) = \frac{\pi}{30} A \tilde{\rho}$.

Multiple-component many-body DPD is also possible by defining partial local densities, e.g. for component α

$$\tilde{\rho}_i^\alpha = \sum_{j \in \alpha, j \neq i} w^\rho(r_{ij}) \tag{10.43}$$

and generalizing Equation 10.40:

$$\vec{F}_{ij}^C = \left(\frac{\partial \psi_{c(i)}^{ex}(\{\tilde{\rho}^\alpha\}_i)}{\partial \tilde{\rho}_{c(j)}} + \frac{\partial \psi_{c(j)}^{ex}(\{\tilde{\rho}^\alpha\}_j)}{\partial \tilde{\rho}_{c(i)}} \right) w^C(r_{ij}) \frac{\vec{r}_{ij}}{r_{ij}} \tag{10.44}$$

with $c(i)$ as the component to which particle i belongs (e.g. if $i \in \alpha$, $c(i) = \alpha$) and $\{\tilde{\rho}^\alpha\}_i$ as the set of local densities of different components at the position of particle i.

The `manybody_module` includes a routine (`local_density`) to calculate local densities for each species using Equation 10.43: the overall density for each particle can be obtained by a simple sum over all species. The user can modify the routines `manybody_force` and `manybody_potential` to apply their own choices for many-body forces and potentials respectively. Up to five many-body interaction parameters per species pair can be specified in the `CONTROL` file.

The many-body DPD example provided with DL_MESO_DPD produces a van der Waals-like equation of state and can be used to model vapour/liquid interfaces[55]. The potential (excess free energy) per particle is given by:

$$\psi^{ex}(\tilde{\rho}) = \frac{\pi}{30} A_{ij} \overline{\tilde{\rho}} + \frac{\pi r_d^4}{30} B_{ij} \tilde{\rho}^2 \tag{10.45}$$

where $\overline{\tilde{\rho}}$ is equivalent to $\tilde{\rho}$ but with the cutoff set to r_c instead of r_d. The associated pairwise force is equal to

$$\vec{F}_{ij}^C = \left[A_{ij} \left(1 - \frac{r_{ij}}{r_c} \right) + B_{ij} \left(\rho_i + \rho_j \right) \left(1 - \frac{r_{ij}}{r_d} \right) \right] \frac{\vec{r_{ij}}}{r_{ij}} \tag{10.46}$$

In the routine provided, the terms with A_{ij} for both force and potential are calculated as though they are standard DPD. By setting $A_{ij} < 0$ and $B_{ij} > 0$, a vapour/liquid mixture can be modelled and its equation of state for a single component is given as

$$p = \rho k_B T + \alpha A \rho^2 + 2\alpha B r_d^4 \left(\rho^3 - c\rho^2 + d\right)$$

where $\alpha \approx 0.101$, c and d are numerical offsets. Parameters A_{ij} and B_{ij}[3] for each species and interacting pair of species can be specified in the `CONTROL` file using the keyword `MANY`.

10.5 Long-ranged Electrostatic (Coulombic) Potentials

Compared to other interactions in DPD, electrostatic interactions act over considerably longer ranges, which can also include periodic images of the system. The governing equation for finding the electric potential is the Poisson equation, shown here in dimensionless form[12]:

$$\nabla \cdot (p(\vec{r})\nabla\varphi) = -\Gamma\rho \tag{10.47}$$

where φ is the electric potential, ρ the charge density (concentration of cations minus concentration of anions per unit volume), $p(\vec{r})$ the local polarizability relative to a reference medium (e.g. water) and Γ the coupling constant for the reference medium. The latter is given by

$$\Gamma = \frac{e^2}{k_B T \epsilon_0 \epsilon_r r_c}$$

with e as the electron charge, ϵ_0 the dielectric constant of a vacuum and ϵ_r the relative permittivity of the reference medium. For water at room temperature (298K) with N_m molecules per DPD particle, $\Gamma \approx 20.00 N_m^{-\frac{1}{3}}$. Alternatively, the total electrostatic potential energy can be expressed as a sum of Coulombic energies (which also include periodic images), i.e.

$$U = \frac{\Gamma}{4\pi} \sum_i \sum_{j>i} \frac{q_i q_j}{|r_{ij}|} \tag{10.48}$$

One method to determine electrostatic interactions is currently included in DL_MESO_DPD. This can be specified in the `CONTROL` file using the keyword `ELEC`: `etype` is an integer specifying the electrostatic calculation method and a, b, c and d are parameters whose meanings are given in Table 10.3.

Table 10.3: Electrostatic coefficients used in DL_MESO_DPD

`etype`	a	b	c	d
1 (Standard Ewald sum/exponential smearing)	Γ	α	β	-

10.5.1 Standard Ewald sum with exponential charge smearing (`etype=1`)

The method currently used in DL_MESO_DPD to determine the electrostatic potential is an Ewald summation[11]:

$$\begin{aligned} U &\equiv q\varphi \\ &= U^{sr} + U^{lr} + U^{sc} + U^{cc} \end{aligned} \tag{10.49}$$

where U^{sr} is a short-range potential energy term that sums quickly in real space, U^{lr} is a long-range term that sums quickly in Fourier or reciprocal space, U^{sc} is a self-energy correction term and U^{cc} is a correction for systems with a net charge.

While the original form of the short-range electrostatic potential uses point charges, this cannot be used unmodified for DPD simulations: soft beads used in combination with unlike point charges would collapse on top

[3] B_{ij} in this case should not be confused with the interaction distance between components i and j.

of each other, forming infinitely strong ion pairs. The charges are therefore spread out over a finite volume using a smearing charge distribution $f(r)$. The current approach uses a Slater-type distribution, i.e.

$$f(r) = \frac{q}{\pi\lambda^3} \exp\left(-\frac{2r}{\lambda}\right) \tag{10.50}$$

where λ is the decay length of the charge. This gives a potential energy between charged particles i and j of

$$U(r_{ij}) = \frac{\Gamma q_i q_j}{4\pi r_{ij}} \left[1 - (1 + \beta r_{ij}) \exp(-2\beta r_{ij})\right] \tag{10.51}$$

and the corresponding electrostatic force

$$\vec{F}_{ij}^{E}(r_{ij}) = \frac{\Gamma q_i q_j}{4\pi r_{ij}^2} \left[1 - \exp(-2\beta r_{ij}) \left(1 + 2\beta r_{ij}(1 + \beta r_{ij})\right)\right] \frac{\vec{r}_{ij}}{r_{ij}} \tag{10.52}$$

where r_e is the electrostatic short-range (real space) cutoff and $\beta \equiv \frac{r_c}{\lambda}$. For large particle separations these expressions reduce down to the standard expressions for point charges and thus the reciprocal space part of the Ewald sum can be calculated without modification.

The real space terms for the Ewald sum are modifications of the above expressions, evaluated for particle pairs when $r_{ij} < r_e$. The short-range potential energy between particles i and j is given as:

$$U_{ij}^{sr} = \frac{\Gamma q_i q_j}{4\pi r_{ij}} \mathrm{erfc}(\alpha r_{ij}) \left(1 - (1 + \beta r_{ij}) \exp(-2\beta r_{ij})\right) \tag{10.53}$$

and the pairwise force:

$$\vec{F}_{ij}^{E,sr} = \frac{\Gamma q_i q_j}{4\pi r_{ij}^2} \left(\frac{2\alpha r_{ij}}{\sqrt{\pi}} \exp(-\alpha^2 r_{ij}^2) + \mathrm{erfc}(\alpha r_{ij})\right) \left(1 - \exp(-2\beta r_{ij}) \left(1 + 2\beta r_{ij}(1 + \beta r_{ij})\right)\right) \frac{\vec{r}_{ij}}{r_{ij}} \tag{10.54}$$

where α is a convergence parameter that controls the real space contribution, chosen to give negligible contributions beyond the real-space cutoff. If a calculational precision of ϵ is required for the Ewald sum, the required value of α is equal to $\frac{\sqrt{|\ln(\epsilon r_e)|}}{r_e}$.

The long-range term for the Ewald sum requires the reciprocal vector $\vec{k}$, which for an orthogonal simulation box of dimensions $L_x \times L_y \times L_z$ is given by

$$\vec{k} = \begin{bmatrix} \frac{2\pi k_1}{L_x} \\ \frac{2\pi k_2}{L_y} \\ \frac{2\pi k_3}{L_z} \end{bmatrix}$$

where k_1, k_2 and k_3 are integers (positive and negative) from zero to large values specified by the user (in the `CONTROL` file using the keyword `KMAX`) as k_1^{max}, k_2^{max} and k_3^{max} respectively for x-, y- and z-dimensions: the maximum reciprocal vector $\vec{k}_{max}$ using the maximum k values can also be defined.

The long-range potential energy term for the entire system is given by:

$$U_{tot}^{lr} = \frac{\Gamma}{2V} \sum_{\vec{k}\neq 0}^{\vec{k}_{max}} \frac{\exp\left(-\frac{k^2}{4\alpha^2}\right)}{k^2} \left| \sum_j q_j \exp\left(-i\vec{k} \cdot \vec{r}_j\right) \right|^2 \tag{10.55}$$

where i is the imaginary constant ($\sqrt{-1}$). Differentiation of the potential gives the long-range electrostatic force on a particle:

$$\vec{F}_j^{E,lr} = -\frac{\Gamma q_j}{V} \sum_{\vec{k}\neq 0}^{\vec{k}_{max}} i\vec{k} \exp\left(i\vec{k} \cdot \vec{r}_j\right) \frac{\exp\left(-\frac{k^2}{4\alpha^2}\right)}{k^2} \sum_n q_n \exp\left(-i\vec{k} \cdot \vec{r}_n\right) \tag{10.56}$$

The self-energy correction term is constant for a given system for all time steps and is equal to

$$U_{tot}^{sc} = -\frac{\Gamma\alpha}{4\pi^{\frac{3}{2}}} \sum_i q_i^2. \tag{10.57}$$

while the system charge correction is

$$U_{tot}^{cc} = -\frac{\Gamma}{8\alpha^2 V}\left(\sum_i q_i\right)^2 \tag{10.58}$$

No additional forces are required for these terms.

In DL_MESO_DPD, the module `ewald_module` contains the most important routines for calculating electrostatic interactions using the algorithm described above. The real space (short-range) component is calculated (in routine `ewald_real`) using a link cell algorithm with a cutoff radius of r_e; a larger boundary halo than for standard pairwise interactions is therefore required. The reciprocal space (long-range) component is calculated using the scheme described by [46], parallelized by distribution over atomic sites, which requires global summations but is more efficient in terms of memory usage than distribution of $\vec{k}$ vectors. The routine `ewald_reciprocal` also adds the self-interaction and charged system corrections, which are calculated using the routine `elecgen` in `config_module`.

10.6 Bond interactions between particles

Molecules of particles bonded together can be included in calculations using a `MOLECULE` file to define the relative positions of particles and a `FIELD` file to define the properties and topologies of the bonds, angles and dihedrals between them. These data are used in the `start` subroutine to add the specified numbers of molecules into the system before DPD calculations commence and to create tables listing the particles that are included in bonds, angles and dihedrals.

10.6.1 Stretching bonds

DL_MESO_DPD can model four forms of bond potential (and corresponding force) between specified particles, all of which are functions of the distance between them. The available bond forms between particles i and j are as follows:

1. Harmonic (Hookean/Fraenkel) bond:

$$U(r_{ij}) = \frac{\kappa}{2}(r_{ij} - r_0)^2 \tag{10.59}$$

2. Finitely Extendible Non-linear Elastic (FENE) bond:

$$U(r_{ij}) = \begin{cases} -\frac{1}{2}\kappa r_{max}^2 \ln\left[1 - \frac{(r_{ij}-r_0)^2}{r_{max}^2}\right] & r_{ij} < r_0 + r_{max} \\ \infty & r_{ij} \geq r_0 + r_{max} \end{cases} \tag{10.60}$$

3. Marko-Siggia Worm-Like Chain (WLC)[28]:

$$U(r_{ij}) = \begin{cases} \frac{k_B T}{2A_p}\left[\frac{1}{2\left(1-\frac{r_{ij}}{r_{max}}\right)} - \frac{1}{2\left(1+\frac{r_{ij}}{r_{max}}\right)} + \frac{r_{ij}^2}{r_{max}^2}\right] & r_{ij} < r_{max} \\ \infty & r_{ij} \geq r_{max} \end{cases} \tag{10.61}$$

4. Morse potential bond[35]:

$$U(r_{ij}) = D_e\left[1 - \exp\left(-\beta\left(r_{ij} - r_0\right)\right)\right] \tag{10.62}$$

where $\vec{r}_{ij} = \vec{r}_i - \vec{r}_j$, r_0 is an equilibrium bond length, r_{max} the maximum specified bond length or extension, κ is a spring force constant, A_p the persistence length of a wormlike chain, D_e the potential well depth and β the potential 'width'. The coefficients for these bonds as used in DL_MESO_DPD and stated in the `FIELD` file are summarized in Table 10.4.

Table 10.4: Bond coefficients used in DL_MESO_DPD

Bond type	a	b	c	d
1 (Harmonic)	κ	r_0	-	-
2 (FENE)	κ	r_0	r_{max}	-
3 (WLC)	A_p	r_{max}	-	-
4 (Morse)	D_e	r_0	β	-

The force on particle i due to a bond potential is obtained from the general formula:

$$\vec{F}_i = -\frac{1}{r_{ij}}\left[\frac{\partial}{\partial r_{ij}} U\left(r_{ij}\right)\right]\vec{r}_{ij} \tag{10.63}$$

with the force acting on particle j equal to the negative of this, and the virial contribution from the stretching bond given by

$$\mathcal{W} = -\vec{r}_{ij}\cdot\vec{F}_i, \tag{10.64}$$

with only *one* contribution per bond[4].

10.6.2 Bond angles

DL_MESO_DPD includes two methods for modelling potentials and forces between three bonded particles due to the angle formed between them, θ_{ijk}. The potentials are given as follows:

1. Harmonic:

$$U\left(\theta_{ijk}\right) = \frac{\kappa}{2}\left(\theta_{ijk} - \theta_0\right)^2 \tag{10.65}$$

2. Harmonic cosine:

$$U\left(\theta_{ijk}\right) = \frac{\kappa}{2}\left(\cos\theta_{ijk} - \cos\theta_0\right)^2 \tag{10.66}$$

where κ is an angle force constant and θ_0 an equilibrium bond angle. Table 10.5 gives the assignments used by DL_MESO_DPD and the `FIELD` file for the bond angle coefficients, with angles given in degrees and cosines automatically calculated from angles.

Table 10.5: Bond angle coefficients used in DL_MESO_DPD

Angle type	a	b	c	d
1 (Harmonic)	κ	θ_0	-	-
2 (Harmonic cosine)	κ	θ_0	-	-

The angle across particles i, j and k can be determined from the bond vectors $\vec{r}_{ij} = \vec{r}_i - \vec{r}_j$ and $\vec{r}_{kj} = \vec{r}_k - \vec{r}_j$:

$$\theta_{ijk} = \cos^{-1}\left\{\frac{\vec{r}_{ij}\cdot\vec{r}_{kj}}{r_{ij}r_{kj}}\right\} \tag{10.67}$$

The most general form for the bond angle potential is given thus:

$$U\left(\theta_{ijk}, r_{ij}, r_{kj}\right) = A\left(\theta_{ijk}\right) S\left(r_{ij}\right) S\left(r_{kj}\right) S\left(r_{ik}\right) \tag{10.68}$$

[4]This expression is also used for the virial contribution from the standard DPD pairwise forces, i.e. Equation (8.3), again only applying a single contribution per particle pair.

with $A(\theta)$ as a purely angular function and $S(r)$ a screening or truncation function. The force on particle ℓ in dimension α is thus given by:

$$\begin{aligned} F_\ell^\alpha &= -\frac{\partial}{\partial x_\alpha} U(\theta_{ijk}, r_{ij}, r_{kj}) && (10.69)\\ &= -S(r_{ij})\, S(r_{kj})\, S(r_{ik})\, \frac{\partial}{\partial r_\ell^\alpha} A(\theta_{ijk}) \\ &\quad -A(\theta_{ijk})\, S(r_{kj})\, S(r_{ik})\, (\delta_{\ell i} - \delta_{\ell j})\, \frac{r_{ij}^\alpha}{r_{ij}} \frac{\partial}{\partial r_{ij}} S(r_{ij}) \\ &\quad -A(\theta_{ijk})\, S(r_{ij})\, S(r_{ik})\, (\delta_{\ell k} - \delta_{\ell j})\, \frac{r_{kj}^\alpha}{r_{kj}} \frac{\partial}{\partial r_{kj}} S(r_{kj}) \\ &\quad -A(\theta_{ijk})\, S(r_{ij})\, S(r_{kj})\, (\delta_{\ell k} - \delta_{\ell i})\, \frac{r_{ik}^\alpha}{r_{ik}} \frac{\partial}{\partial r_{ik}} S(r_{ik}) && (10.70) \end{aligned}$$

with $\delta_{ab} = 1$ if $a = b$ and $\delta_{ab} = 0$ if $a \neq b$. In the absence of screening terms, the above formula reduces to

$$\begin{aligned} F_\ell^\alpha &= -\frac{\partial}{\partial r_\ell^\alpha} A(\theta_{ijk}) && (10.71)\\ &= \frac{1}{\sin\theta_{ijk}} \frac{\partial}{\partial \theta_{ijk}} A(\theta_{ijk}) \times \\ &\quad \left\{ (\delta_{\ell j} - \delta_{\ell i}) \frac{r_{kj}^\alpha}{r_{ij} r_{kj}} + (\delta_{\ell j} - \delta_{\ell k}) \frac{r_{ij}^\alpha}{r_{ij} r_{kj}} - \cos(\theta_{ijk}) \left[(\delta_{\ell j} - \delta_{\ell i}) \frac{r_{ij}^\alpha}{r_{ij}^2} + (\delta_{\ell j} - \delta_{\ell k}) \frac{r_{kj}^\alpha}{r_{kj}^2} \right] \right\} && (10.72) \end{aligned}$$

The contribution to the virial from the angle is given by

$$\mathcal{W} = -\left(\vec{r}_{ij} \cdot \vec{F}_i + \vec{r}_{kj} \cdot \vec{F}_k \right) \tag{10.73}$$

which is equal to zero for bond angle potentials without screening terms[47].

10.6.3 Bond dihedrals

Three potential models for bond dihedrals along particles i, j, k and l are provided in DL_MESO_DPD as follows:

1. Cosine torsion:
$$U(\phi_{ijkl}) = A\left[1 + \cos(m\phi_{ijkl} - \delta)\right] \tag{10.74}$$
2. Harmonic:
$$U(\phi_{ijkl}) = \frac{\kappa}{2} (\phi_{ijkl} - \phi_0)^2 \tag{10.75}$$
3. Harmonic cosine:
$$U(\phi_{ijkl}) = \frac{\kappa}{2} (\cos\phi_{ijkl} - \cos\phi_0)^2 \tag{10.76}$$

where A and κ are dihedral force constants, m is the multiplicity, δ the dihedral at minimum potential and ϕ_0 an equilibrium bond dihedral. The assignment of coefficients used in DL_MESO_DPD and the `FIELD` file is given by Table 10.6.

Table 10.6: Bond dihedral coefficients used in DL_MESO_DPD

Dihedral type	a	b	c	d
1 (Cosine torsion)	A	m	δ	-
2 (Harmonic)	κ	ϕ_0	-	-
3 (Harmonic cosine)	κ	ϕ_0	-	-

The dihedral angle across all four particles (or between planes ij and kl) is given by

$$\phi_{ijkl} = \cos^{-1} B(\vec{r}_{ij}, \vec{r}_{jk}, \vec{r}_{kl}) \tag{10.77}$$

where

$$B\left(\vec{r}_{ij},\vec{r}_{jk},\vec{r}_{kl}\right)=\frac{\left(\vec{r}_{ij}\times\vec{r}_{jk}\right)\cdot\left(\vec{r}_{jk}\times\vec{r}_{kl}\right)}{|\vec{r}_{ij}\times\vec{r}_{jk}||\vec{r}_{jk}\times\vec{r}_{kl}|} \tag{10.78}$$

which gives a negative value for ϕ_{ijkl} if the vector $\left(\vec{r}_{ij}\times\vec{r}_{jk}\right)\cdot\left(\vec{r}_{jk}\times\vec{r}_{kl}\right)$ is in the same direction as the bond vector $\vec{r}_{jk}$ and positive if in the opposite direction.

The force on particle ℓ acting in the α direction is given by

$$\begin{aligned} F_{\ell}^{\alpha} &= -\frac{\partial}{\partial x_{\alpha}}U\left(\phi_{ijkl}\right) && (10.79)\\ &= \frac{1}{\sin\phi_{ijkl}}\frac{\partial}{\partial\phi_{ijkl}}U\left(\phi_{ijkl}\right)\frac{\partial}{\partial r_{\ell}^{\alpha}}B\left(\vec{r}_{ij},\vec{r}_{jk},\vec{r}_{kl}\right). && (10.80)\end{aligned}$$

Using the following definition

$$\left[\vec{a}\ \vec{b}\right]_{\alpha}\equiv\sum_{\beta}\left(1-\delta_{\alpha\beta}\right)a^{\beta}b^{\beta}$$

the derivative of $B\left(\vec{r}_{ij},\vec{r}_{jk},\vec{r}_{kl}\right)$ is given by

$$\begin{aligned}\frac{\partial}{\partial r_{\ell}^{\alpha}}B\left(\vec{r}_{ij},\vec{r}_{jk},\vec{r}_{kl}\right)&=\frac{1}{|\vec{r}_{ij}\times\vec{r}_{jk}||\vec{r}_{jk}\times\vec{r}_{kl}|}\frac{\partial}{\partial r_{\ell}^{\alpha}}\left\{\left(\vec{r}_{ij}\times\vec{r}_{jk}\right)\cdot\left(\vec{r}_{jk}\times\vec{r}_{kl}\right)\right\}\\ &-\frac{\cos\phi_{ijkl}}{2}\left\{\frac{1}{|\vec{r}_{ij}\times\vec{r}_{jk}|^{2}}\frac{\partial}{\partial r_{\ell}^{\alpha}}|\vec{r}_{ij}\times\vec{r}_{jk}|^{2}+\frac{1}{|\vec{r}_{jk}\times\vec{r}_{kl}|^{2}}\frac{\partial}{\partial r_{\ell}^{\alpha}}|\vec{r}_{jk}\times\vec{r}_{kl}|^{2}\right\} \qquad (10.81)\end{aligned}$$

with

$$\begin{aligned}\frac{\partial}{\partial r_{\ell}^{\alpha}}\left\{\left(\vec{r}_{ij}\times\vec{r}_{jk}\right)\cdot\left(\vec{r}_{jk}\times\vec{r}_{kl}\right)\right\} &= r_{ij}^{\alpha}\left(\left[\vec{r}_{jk}\vec{r}_{jk}\right]_{\alpha}\left(\delta_{\ell k}-\delta_{\ell l}\right)+\left[\vec{r}_{jk}\vec{r}_{kl}\right]_{\alpha}\left(\delta_{\ell k}-\delta_{\ell j}\right)\right)+\\ &\quad r_{jk}^{\alpha}\left(\left[\vec{r}_{ij}\vec{r}_{jk}\right]_{\alpha}\left(\delta_{\ell l}-\delta_{\ell k}\right)+\left[\vec{r}_{jk}\vec{r}_{kl}\right]_{\alpha}\left(\delta_{\ell j}-\delta_{\ell i}\right)\right)+\\ &\quad r_{kl}^{\alpha}\left(\left[\vec{r}_{ij}\vec{r}_{jk}\right]_{\alpha}\left(\delta_{\ell k}-\delta_{\ell j}\right)+\left[\vec{r}_{jk}\vec{r}_{jk}\right]_{\alpha}\left(\delta_{\ell i}-\delta_{\ell j}\right)\right)+\\ &\quad 2r_{jk}^{\alpha}\left[\vec{r}_{ij}\vec{r}_{kl}\right]_{\alpha}\left(\delta_{\ell l}-\delta_{\ell k}\right) \qquad (10.82)\end{aligned}$$

$$\begin{aligned}\frac{\partial}{\partial r_{\ell}^{\alpha}}|\vec{r}_{ij}\times\vec{r}_{jk}|^{2} &= 2r_{ij}^{\alpha}\left(\left[\vec{r}_{jk}\vec{r}_{jk}\right]_{\alpha}\left(\delta_{\ell j}-\delta_{\ell i}\right)+\left[\vec{r}_{ij}\vec{r}_{jk}\right]_{\alpha}\left(\delta_{\ell j}-\delta_{\ell k}\right)\right)+\\ &\quad 2r_{jk}^{\alpha}\left(\left[\vec{r}_{ij}\vec{r}_{ij}\right]_{\alpha}\left(\delta_{\ell k}-\delta_{\ell j}\right)+\left[\vec{r}_{ij}\vec{r}_{jk}\right]_{\alpha}\left(\delta_{\ell i}-\delta_{\ell j}\right)\right) \qquad (10.83)\end{aligned}$$

$$\begin{aligned}\frac{\partial}{\partial r_{\ell}^{\alpha}}|\vec{r}_{jk}\times\vec{r}_{kl}|^{2} &= 2r_{kl}^{\alpha}\left(\left[\vec{r}_{jk}\vec{r}_{jk}\right]_{\alpha}\left(\delta_{\ell l}-\delta_{\ell k}\right)+\left[\vec{r}_{jk}\vec{r}_{kl}\right]_{\alpha}\left(\delta_{\ell j}-\delta_{\ell k}\right)\right)+\\ &\quad 2r_{jk}^{\alpha}\left(\left[\vec{r}_{kl}\vec{r}_{kl}\right]_{\alpha}\left(\delta_{\ell k}-\delta_{\ell j}\right)+\left[\vec{r}_{jk}\vec{r}_{kl}\right]_{\alpha}\left(\delta_{\ell k}-\delta_{\ell l}\right)\right) \qquad (10.84)\end{aligned}$$

It can be shown both algebraically and thermodynamically that the dihedral makes no contribution to the virial[47].

Improper dihedrals — which limit the geometry of molecules — can be applied using the same procedure as standard dihedrals and no distinction is made between them in DL_MESO_DPD.

10.7 Surface interactions

The default boundaries for a simulation box are periodic, i.e. particles leaving the system are replaced at the opposite face with the same velocity. Certain systems may require alternative boundary conditions and DL_MESO_DPD can include these at the system boundaries. The keyword `SURFACE` in the `CONTROL` file can be used to specify the type of surface interaction (`srftype`) and which surface(s) are to be included (`srfx`, `srfy`, `srfz`: each set to 0 for periodic boundaries and 1 or greater for other types), as well as any necessary parameters. The module `surface_module` includes routines to apply boundary conditions at those surfaces and to identify the processing units that include surfaces (`surfacenodes`).

10.7.1 Hard reflecting boundaries (`srftype=1`)

This boundary condition is applied by using a combination of specular (free-slip) reflection at the system boundaries and a soft short-range wall repulsion to reduce density oscillations[56]. The reflection is achieved by moving any particle leaving the system at a particular boundary back into it and inverting the velocity component normal to the wall (but maintaining the tangential momentum), which is achieved with the routine `surfacebounce`. The soft short-range wall repulsion is given by

$$U_{wall}(z) = \tfrac{1}{2} A_{wall,\alpha} \left(\frac{1-z}{z_c} \right)^2, \qquad (z < z_c) \tag{10.85}$$

where $A_{wall,\alpha}$ is the repulsive force magnitude with species α, z is the distance between the particle and the wall and z_c the surface repulsion range. The repulsion is applied at the same time as all pairwise forces using the routine `hardreflect`. Since the walls are assumed to be non-porous, no interactions across them are included and thus boundary halos adjacent to walls are not used.

The repulsive force magnitudes between the boundary and each species are specified in the `CONTROL` file at the end of the line with the keyword `SURFACE` in order of definition.

Chapter 11

DL_MESO_DPD Package Reference

11.1 Overview

DL_MESO_DPD consists of seventeen Fortran90 modules, which should be compiled in the following order prior to the main program itself:

- `constants`
  Contains the constants and parameters required by DL_MESO_DPD.
- `variables`
  Contains the variables and arrays required by DL_MESO_DPD.
- `numeric_container`
  Contains random number generators and a general-purpose scalar sum function.
- `comms_module`
  Contains all subroutines necessary for parallel computation.
- `error_module`
  Contains subroutines to print error messages and close down DL_MESO_DPD in a controlled manner.
- `parse_utils`
  Contains functions to read in text data from input files.
- `surface_module`
  Contains subroutines for applying boundary conditions at system planes, e.g. solid walls.
- `ewald_module`
  Contains subroutines for calculating forces due to electrostatic interactions using Ewald summation-based methods.
- `manybody_module`
  Contains subroutines for calculating density-dependent forces between particles, including the calculation of localized densities.
- `bond_module`
  Contains book-keeping and force calculation routines for bonds, angles and dihedrals.
- `domain_module`
  Contains subroutines to construct parallel link cells, import and deport particles in and export particle data to domain boundary halos.

- `start_module`
  Contains subroutines to initialize and restart DPD calculations.
- `config_module`
  Contains subroutines to read in input files with system and molecule/bond data, to zero all parameters and accumulators for statistical data and (for parallel version only) determine 3D domain decomposition.
- `field_module`
  Contains subroutines to calculate pairwise forces between particles.
- `integrate_module`
  Contains subroutines to integrate the equations of motion using the Velocity Verlet scheme and apply various thermostats and barostats.
- `statistics_module`
  Contains subroutines to calculate and write out statistical and trajectory data.
- `run_module`
  Contains program loops for different integrators and barostats.

Most of the above modules and the main program file `dlmesodpd.f90` are identical for both serial and parallel versions of DL_MESO_DPD. The filenames for the serial versions of `comms_module` and `domain_module` end with `_ser`, but are referred to in the code by their standard names.

The modules `manybody_module` and `bond_module` may be modified by users to incorporate alternative many-body DPD schemes or additional bond, angle and dihedral models. For anything else, however, we recommend that DL_MESO users put any self-defined subroutines and functions into a module file called `user_module.f90` so future upgrades of DL_MESO will not interfere with their contributions.

11.2 DL_MESO_DPD Subroutines and Functions

11.2.1 Main program: dlmesodpd

Both the serial and parallel versions of the program operate in a similar way. Before DPD calculations start, the following tasks are carried out:

- For parallel running only, MPI is started up (`initcomms`) and the node properties are determined.
- An I/O channel for the general `OUTPUT` file is opened.
- The system clock is consulted for a start time (`timchk`).
- The starting banner for DL_MESO_DPD is printed.
- Initial values are set (`zero`).
- System and bond data are read in and initialized (`sysdef`).
- The starting configuration is set up (`start`), either from scratch or specified by the user using a `CONFIG` file.
- The system clock is consulted again for the start of the DPD calculation cycle (`timchk`).

A loop for DPD calculations is then called from `run_module` depending on the integrator and barostat selected by the user. Each step of DPD calculations involves the following:

- The step counter `nstep` is increased.
- The system clock is checked (`timchk`).
- The first stage of the required integrator is used to update the motion of the particles and their positions.
- Masses and particle/molecule names are reassigned to the particles.
- The parallel link-cell structure is set up and the pairwise forces calculated (`plcfor_*`).
- The time taken to calculate the forces is determined (`timchk`).
- The second stage of the integrator is applied to calculate the velocities at the end of the time step. This may include e.g. recalculation of dissipative forces or resizing of the system for a barostat.
- Statistical properties for the system are calculated and, during equilibration, the particle velocities are rescaled for the specified temperature (`statis`).
- After every `nsbpo` time steps, the system clock is consulted (`timchk`) and statistical data is printed to the `OUTPUT` file (`printout`). If equilibration has come to an end, i.e. `nstep = nseql`, this is also reported (`equilout`).
- If requested by the user, after every `iscorr` time steps statistical data is written to the `CORREL` file (`corout`) and after every `nsbsv` time steps trajectory data is saved to the `HISTORY` or `HISTORY*` file(s).
- The step time is calculated (`timchk`) and if the allocated time has expired, the job is closed down; otherwise restart data is saved (`revive`).

After all the time steps have been calculated or the allocated time has elapsed:

- The final calculation summary is printed to the `OUTPUT` file (`result`).
- The duration of the calculation run is determined and printed (`timchk`).
- All remaining output channels (for `OUTPUT`, `HISTORY*` files) are closed.
- For parallel running only, MPI communications are closed down (`exitcomms`).

11.2.2 numeric_container

This package contains general purpose functions which may be replaced with any suitable functions in Fortran90 standard libraries, as well as bookkeeping subroutines for the global/local particle number list.

duni

- **Header records**
  `REAL(KIND=dp) FUNCTION duni (idnode)`
- **Function**
  Creates a double precision random number between 0 and 1.
- **Dependencies**
  None
- **Arguments**

`idnode`	input	integer
`duni`	output	real(KIND=dp)

- **Comments**
  The random number generator is an implementation of the Universal Random Number Generator[30]. The processor name `idnode` is used as a seed, which is activated the first time the function is called.

mtrnd

- **Header records**
  `REAL(KIND=dp) FUNCTION mtrnd (idnode)`
- **Function**
  Creates a double precision random number between 0 and 1.
- **Dependencies**
  None
- **Arguments**

`idnode`	input	integer
`mtrnd`	output	real(KIND=dp)

- **Comments**
  The random number generator is an implementation of the Mersenne Twister random number generator[33]. The processor name `idnode` is used as a seed, which is activated the first time the function is called.

gaussmp

- **Header records**
  `REAL(KIND=dp) FUNCTION gaussmp (idnode)`
- **Function**
  Creates a Gaussian random number.
- **Dependencies**
  `mtrnd`
- **Arguments**

`idnode`	input	integer
`gaussmp`	output	real(KIND=dp)

- **Comments**
  This is an implementation of the Marsaglia polar method[29] to convert linear random numbers (generated by the Mersenne Twister method) to Gaussian random variables with zero mean and unity variance.

sclsum

- **Header records**
  `REAL(KIND=dp) FUNCTION sclsum (n, a, i)`
- **Function**
  Calculates the scalar sum of an array.
- **Dependencies**
  None
- **Arguments**

`n`	input	integer
`a`	input	array of real(KIND=dp)
`i`	input	integer
`sclsum`	output	real(KIND=dp)

erfcdp

- **Header records**
  `REAL(KIND=dp) FUNCTION erfcdp (x)`
- **Function**
  Calculates the complementary error function for `x`, erfc(x).
- **Dependencies**
  None
- **Arguments**

`x`	input	real(KIND=dp)
`erfcdp`	output	real(KIND=dp)

- **Comments**
  This approximation for the function is based on a Chebyshev polynomial fitting[15].

11.2.3 comms_module

This module is essential for parallel running and does not require detailed knowledge for its use: depending on the version and implementation of MPI available, the user may wish to select between the lines `USE MPI` and `INCLUDE "mpif.h"` for loading the necessary routines. The serial version of the `comms_module` primarily consists of dummy routines to satisfy the required calls in the rest of the code.

initcomms

- **Header records**
  `SUBROUTINE initcomms ()`
- **Function**
  Starts Message Passing Interface (MPI).
- **Dependencies**
  None

exitcomms

- **Header records**
  `SUBROUTINE exitcomms ()`
- **Function**
  Closes Message Passing Interface (MPI) in a controlled manner.
- **Dependencies**
  None

abortcomms

- **Header records**
  `SUBROUTINE abortcomms ()`
- **Function**
  Terminates Message Passing Interface (MPI).
- **Dependencies**
  None

gsync

- **Header records**
  `SUBROUTINE gsync ()`
- **Function**
  Pauses running until all processes are synchronized.
- **Dependencies**
  None

global_and

- **Header records**
  `SUBROUTINE global_and (iii, nnn)`
- **Function**
  Finds global logical AND from a Boolean array `iii()` of size `nnn`.
- **Dependencies**
  None
- **Arguments**

`iii`	input/output	array of logical
`nnn`	input	integer

- **Comments**
  For Boolean scalars, the alternative `SUBROUTINE global_sca_and (iii)` is available.

global_sum_dble

- **Header records**
  `SUBROUTINE global_sum_dble (aaa, nnn)`
- **Function**
  Globally sums double precision array `aaa()` of size `nnn`.
- **Dependencies**
  None
- **Arguments**

`aaa`	input/output	array of real(KIND=dp)
`nnn`	input	integer

- **Comments**
  For double precision scalars, the alternative `SUBROUTINE global_sca_sum_dble (aaa)` is available.

global_sum_int

- **Header records**
  `SUBROUTINE global_sum_int (iii, nnn)`
- **Function**
  Globally sums integer array `iii()` of size `nnn`.
- **Dependencies**
  None

- **Arguments**

`iii`	input/output	array of integers
`nnn`	input	integer

- **Comments**
  For integer scalars, the alternative `SUBROUTINE global_sca_sum_int (iii)` is available.

msg_receive_blocked

- **Header records**
  `SUBROUTINE msg_receive_blocked (msgtag, buf, length)`
- **Function**
  In a blocking call, receives data in the form of a double precision array `buf()`.
- **Dependencies**
  None
- **Arguments**

`buf`	output	array of real(KIND=dp)
`msgtag`	input	integer
`length`	input	integer

- **Comments**
  For a double precision scalar, the alternative `SUBROUTINE msg_receive_sca_blocked (msgtag, buf, length)` is available.

msg_receive_unblocked

- **Header records**
  `INTEGER FUNCTION msg_receive_unblocked (msgtag, buf, length)`
- **Function**
  In a non-blocking call, receives data in the form of a double precision array `buf()`.
- **Dependencies**
  None
- **Arguments**

`buf`	output	array of real(KIND=dp)
`msgtag`	input	integer
`length`	input	integer
`msg_receive_unblocked`	output	integer

- **Comments**
  Since no scalars are received in non-blocking calls, no scalar version of this function exists.

msg_send_blocked

- **Header records**
  `SUBROUTINE msg_send_blocked (msgtag, buf, length, pe)`
- **Function**
  In a blocking call, send data from a double precision array `buf()`.
- **Dependencies**
  None

- **Arguments**

`buf`	input	array of real(KIND=dp)
`msgtag`	input	integer
`length`	input	integer
`pe`	input	integer

- **Comments**
  For a double precision scalar, the alternative `SUBROUTINE msg_send_sca_blocked (msgtag, buf, length, pe)` is available.

msg_wait

- **Header records**
  `SUBROUTINE msg_wait (request)`
- **Function**
  Causes process to wait for an unblocked message.
- **Dependencies**
  None
- **Arguments**

`request`	input	integer

mynode

- **Header records**
  `INTEGER FUNCTION mynode ()`
- **Function**
  Returns name of process.
- **Dependencies**
  None
- **Arguments**

`mynode`	output	integer

- **Comments**
  The serial version of this function returns 0.

numnodes

- **Header records**
  `INTEGER FUNCTION numnodes ()`
- **Function**
  Returns total number of processes available.
- **Dependencies**
  None
- **Arguments**

`numnodes`	output	integer

- **Comments**
  The serial version of this function returns 1.

timchk

- **Header records**
  `SUBROUTINE timchk(ktim, time)`
- **Function**
  Determines the time elapsed since the start of the calculation run and, if `ktim` > 0, prints the time to the `OUTPUT` file.
- **Dependencies**
  None
- **Arguments**

`ktim`	input	integer
`time`	output	real(KIND=dp)

- **Comments**
  The serial version of DL_MESO_DPD uses the generic `SYSTEM_CLOCK` call, while the parallel version uses `MPI_wtime`.

11.2.4 error_module

This module is used to print error messages and shut down DL_MESO_DPD in a controlled manner. It requires the modules `constants` and `comms_module` to be loaded beforehand.

error

- **Header records**
  `SUBROUTINE error (idnode, iode, value)`
- **Function**
  Prints user-friendly error message in OUTPUT file and closes down DL_MESO_DPD.
- **Dependencies**
  `abortcomms`
- **Arguments**

`idnode`	input	integer
`iode`	input	integer
`value`	input	integer

- **Comments**
  The error code `iode` closes down DL_MESO_DPD when positive; negative values can be used to print warning messages for non-fatal problems.

11.2.5 parse_utils

getword

- **Header records**
  `CHARACTER(16) FUNCTION getword (txt, n)`
- **Function**
  Obtains the `n`th word from a line of text `txt` separated by spaces or commas.

- **Dependencies**
  None

- **Arguments**

`txt`	input	character(160)
`n`	input	integer
`getword`	output	character(16)

parseint

- **Header records**
  `INTEGER(KIND=li) FUNCTION parseint (word)`

- **Function**
  Reads the integer contained in the string `word`.

- **Dependencies**
  None

- **Arguments**

`word`	input	character(16)
`parseint`	output	integer(KIND=li)

parsedble

- **Header records**
  `REAL(KIND=dp) FUNCTION parsedble (word)`

- **Function**
  Reads the double precision number contained in the string `word`.

- **Dependencies**
  None

- **Arguments**

`word`	input	character(16)
`parsedble`	output	real(KIND=dp)

getint

- **Header records**
  `INTEGER(KIND=li) FUNCTION getint (txt, n)`

- **Function**
  Reads the `n`th 'word' of the string `txt` to obtain an integer.

- **Dependencies**
  `parseint`

- **Arguments**

`txt`	input	character(160)
`n`	input	integer
`getint`	output	integer(KIND=li)

getdble

- **Header records**
  `REAL(KIND=dp) FUNCTION getdble (txt, n)`
- **Function**
  Reads the `n`th 'word' of the string `txt` to obtain a double precision number.
- **Dependencies**
  `parsedble`
- **Arguments**

`txt`	input	character(160)
`n`	input	integer
`getdble`	output	real(KIND=dp)

11.2.6 surface_module

This module requires the `variables` module to be loaded beforehand.

surfacenodes

- **Header records**
  `SUBROUTINE surfacenodes`
- **Function**
  Identifies nodes containing surfaces or other boundary conditions.
- **Dependencies**
  None

hardreflect

- **Header records**
  `SUBROUTINE hardreflect (k)`
- **Function**
  Applies boundary condition for hard reflecting walls: calculates short-range forces on particles when `k=1`, applies bounce-back condition when `k=2` for particles about to pass through boundary.
- **Dependencies**
  `surfacebounce`
- **Arguments**

`k`	input	integer

- **Comments**
  The boundary condition is given by [56]: the applied wall potential on each particle is given as

$$U_{wall,i}(z) = \frac{1}{2} A_{wall,i} \left(\frac{1-z}{z_c} \right)^2$$

  for $z < z_c$.

surfacebounce

- **Header records**
  `SUBROUTINE surfacebounce (ddd, vdd, sided)`
- **Function**
  Applies boundary condition on a leaving particle by means of specular reflection: move particle back into system and invert velocity component normal to wall.
- **Dependencies**
  None
- **Arguments**

`ddd`	input/output	real(KIND=dp)
`vdd`	input/output	real(KIND=dp)
`sided`	input	real(KIND=dp)

- **Comments**
  Momentum tangential to the boundary is preserved, i.e. this applies a free-slip boundary condition.

11.2.7 ewald_module

This module requires the `constant`, `variables`, `numeric_container` and `comms_module` modules to be loaded beforehand.

ewald_real

- **Header records**
  `SUBROUTINE ewald_real (nlimit)`
- **Function**
  Calculates real-space terms for Ewald summation with Slater-type (exponential decay) charge distributions.
- **Dependencies**
  `erfcdp`
- **Arguments**

`nlimit`	input	integer

- **Comments**
  Calculates short-range Coulombic forces and potential energies for Ewald summation using the following smeared (non-point) charge distribution[11]:

$$f(r) = \frac{q\beta^3}{\pi r_c^3} \exp\left(-\frac{2r\beta}{r_c}\right)$$

ewald_reciprocal

- **Header records**
  `SUBROUTINE ewald_reciprocal`
- **Function**
  Calculates reciprocal-space terms for Ewald summation.
- **Dependencies**
  None

- **Comments**
  Calculates long-range Coulombic forces and potential energies using standard Ewald summation, including self-energy corrections for charged systems.

11.2.8 manybody_module

This module requires pre-loading of the `constants` and `variables` modules.

local_density

- **Header records**
  `SUBROUTINE local_density`
- **Function**
  Calculates local densities for many-body DPD interactions.
- **Dependencies**
  `weight_rho`
- **Comments**
  Uses the parallel link-cell structure to calculate local densities for each component,

$$\rho_i = \sum_{j \neq i} w^{\rho}(r_{ij})$$

  omitting self-contributions for each particle[53].

weight_rho

- **Header records**
  `REAL(KIND=dp) FUNCTION weight_rho (rrr)`
- **Function**
  Calculates normalized weight function for local densities.
- **Dependencies**
  None
- **Comments**
  The default weight function[55] is

$$w^{\rho}(r) = \frac{15}{2\pi r_d^3}\left(1 - \frac{r}{r_d}\right)^2,$$

  which may be changed by the user.

manybody_force

- **Header records**
  `SUBROUTINE manybody_force (i, j, k, rrr, mbforce)`
- **Function**
  Calculates many-body DPD interaction force and non-density-dependent potential energies between particles `i` and `j` using parameter set `k` and inter-particle distance `rrr`.
- **Dependencies**
  None

- **Arguments**

`i`	input	integer
`j`	input	integer
`k`	input	integer
`rrr`	input	real(KIND=dp)
`mbforce`	output	real(KIND=dp)

- **Comments**
  The default form for the many-body DPD interaction force is a two-term style suitable for modelling vapour-liquid mixtures[55]:

$$\vec{F}_{ij}^{C} = \left[A_{ij} \left(1 - \frac{r_{ij}}{r_c} \right) + B_{ij}(\rho_i + \rho_j) \left(1 - \frac{r_{ij}}{r_d} \right) \right] \frac{\vec{r}_{ij}}{r_{ij}}.$$

  This subroutine may be changed by users who wish to use different many-body interaction functional forms.

manybody_potential

- **Header records**
  `SUBROUTINE manybody_potential`

- **Function**
  Calculates the self-energy for every particle resulting from density-dependent (many-body) interactions.

- **Dependencies**
  None

- **Comments**
  The default form is based on the two-term vapour-liquid interactions used for many-body forces[55], although only the density-dependent potential energies are calculated using this routine (the rest are calculated in `manybody_force`). Users may wish to modify this routine to use their own many-body interaction models.

11.2.9 bond_module

This module requires the modules `constants`, `variables`, `error_module` and `comms_module` to be loaded beforehand.

shellsort_list

- **Header records**
  `SUBROUTINE shellsort_list`

- **Function**
  Reorders the global/local particle number list (in terms of global particle number) using a Shell sort.

- **Dependencies**
  None

search_list

- **Header records**
  `INTEGER FUNCTION search_list (aim)`

- **Function**
  Determines the index for the global/local particle number list for a specified global particle number `aim` using a binary search.

- **Dependencies**
  None

- **Arguments**

`aim`	input	integer
`search_list`	output	integer

- **Comments**
  This function returns a negative value if the global particle number cannot be found in the list. If it is not the only entry for the global particle number, the function returns the index plus the value of `nlist` (number of list items) to flag up duplicate entries.

duplicate

- **Header records**
  `SUBROUTINE duplicate (global1, global2, ind1, ind2)`

- **Function**
  Determines the indices out of duplicate entries from the global/local particle number list for a pair of specified global particle numbers, `global1` and `global2`, that produce the shortest distance between the particles.

- **Dependencies**
  None

- **Arguments**

`global1`	input	integer
`global2`	input	integer
`ind1`	input/output	integer
`ind2`	input/output	integer

contract_bndtbl

- **Header records**
  `SUBROUTINE contract_bndtbl`

- **Function**
  Strips out all bond pairs from bond table that have been reassigned to neighbouring processors.

- **Dependencies**
  None

- **Comments**
  Only called for parallel version of DL_MESO_DPD when bond tables include only local bonds in each processor.

contract_angtbl

- **Header records**
  `SUBROUTINE contract_angtbl`

- **Function**
  Strips out all bond angle triples from angle table that have been reassigned to neighbouring processors.
- **Dependencies**
  None
- **Comments**
  Only called for parallel version of DL_MESO_DPD when angle tables include only local angles in each processor.

contract_dhdtbl

- **Header records**
  `SUBROUTINE contract_dhdtbl`
- **Function**
  Strips out all bond dihedral quadruples from dihedral table that have been reassigned to neighbouring processors.
- **Dependencies**
  None
- **Comments**
  Only called for parallel version of DL_MESO_DPD when dihedral tables include only local dihedrals in each processor.

bond_force

- **Header records**
  `SUBROUTINE bond_force (bondtype, r, a, b, c, d, force, potential)`
- **Function**
  Determines the stretching force and potential energy between a pair of bonded particles.
- **Dependencies**
  None
- **Arguments**

`bondtype`	input	integer
`r`	input	real(KIND=dp)
`a`	input	real(KIND=dp)
`b`	input	real(KIND=dp)
`c`	input	real(KIND=dp)
`d`	input	real(KIND=dp)
`force`	output	real(KIND=dp)
`potential`	output	real(KIND=dp)

- **Comments**
  If required, the user can add extra stretching bond interaction types in this subroutine as additional cases.

angle_force

- **Header records**
  `SUBROUTINE angle_force (angtype, theta, rab, rcb, a, b, c, d, force, potential, virial, dfab, dfcb)`

- **Function**
  Determines the bond angle force, potential energy and virial across a triple of bonded particles.
- **Dependencies**
  None
- **Arguments**

`angtype`	input	integer
`theta`	input	real(KIND=dp)
`rab`	input	real(KIND=dp)
`rcb`	input	real(KIND=dp)
`a`	input	real(KIND=dp)
`b`	input	real(KIND=dp)
`c`	input	real(KIND=dp)
`d`	input	real(KIND=dp)
`force`	output	real(KIND=dp)
`potential`	output	real(KIND=dp)
`virial`	output	real(KIND=dp)
`dfab`	output	real(KIND=dp)
`dfcb`	output	real(KIND=dp)

- **Comments**
  If required, the user can add extra bond angle interaction types in this subroutine as additional cases.

dihedral_force

- **Header records**
  `SUBROUTINE dihedral_force (dhdtype, phi, pb, pc, a, b, c, d, force, potential)`
- **Function**
  Determines the bond dihedral force and potential energy across a quadruple of bonded particles.
- **Dependencies**
  None
- **Arguments**

`dhdtype`	input	integer
`phi`	input	real(KIND=dp)
`pb`	input	real(KIND=dp)
`pc`	input	real(KIND=dp)
`a`	input	real(KIND=dp)
`b`	input	real(KIND=dp)
`c`	input	real(KIND=dp)
`d`	input	real(KIND=dp)
`force`	output	real(KIND=dp)
`potential`	output	real(KIND=dp)

- **Comments**
  If required, the user can add extra bond dihedral (and improper dihedral) interaction types in this subroutine as additional cases.

bondforceslocal

- **Header records**
  `SUBROUTINE bondforceslocal`

- **Function**
  Calculates all bond (stretching, angle, dihedral) forces between particles in system using locally-defined bond lists.

- **Dependencies**
  `shellsort_list`
  `search_list`
  `duplicate`
  `error`
  `global_sca_and`
  `bond_force`
  `angle_force`
  `dihedral_force`

- **Comments**
  Assumes that bond lists contain only those bonds in the current node. This is the most efficient method for parallel running but runs the risk of losing track of bonded pairs if bond lengths get longer than the subdomain halo size `rhalo`.

bondforcesglobal

- **Header records**
  `SUBROUTINE bondforcesglobal`

- **Function**
  Calculates all bond (stretching, angle, dihedral) forces between particles in system using globally-defined bond lists.

- **Dependencies**
  `shellsort_list`
  `search_list`
  `duplicate`
  `error`
  `global_sum_dble`
  `global_sca_and`
  `bond_force`
  `angle_force`
  `dihedral_force`

- **Comments**
  Assumes that bond lists contain all bonds in the entire system. This is less efficient for parallel running than only calculating the bonds in each node but ensures longer bond lengths can be accommodated.

11.2.10 domain_module

This module requires the modules `constants`, `variables`, `error_module`, `comms_module` (if running parallel version) and `bond_module` (if running parallel version) to be loaded beforehand.

domain

- **Header records**
  `SUBROUTINE domain`

- **Function**
  Determines 3D domain decomposition for system: number of nodes in each direction, location for each node in system, nearest neighbouring nodes.

- **Dependencies** (parallel version only)
  `msg_receive_blocked`
  `msg_receive_sca_blocked`
  `msg_send_blocked`
  `msg_send_sca_blocked`

- **Comments**
  Essential for parallel version of DL_MESO_DPD; the serial version of this subroutine sets values for a single processor.

parlnk

- **Header records**
  `SUBROUTINE parlnk (num1, num2)`

- **Function**
  Constructs the parallel link cells for calculations of pairwise forces between particles.

- **Dependencies**
  None

- **Arguments**

`num1`	input	integer
`num2`	input	integer

- **Comments**
  Pairwise electrostatic forces typically act over longer lengthscales and are not considered using this subroutine: a similar routine for the real-space part of Ewald summation is included in `ewald_real`.

deport

- **Header records**
  `SUBROUTINE deport (nlimit, mdir, mp, begin, final, shove)`

- **Function**
  Deports particles from boundary halo to neighbouring domain.

- **Dependencies**
  `contract_bndtbl`
  `contract_angtbl`
  `contract_dhdtbl`
  `error`
  `global_sca_and`
  `msg_receive_unblocked`
  `msg_receive_sca_blocked`
  `msg_send_blocked`
  `msg_send_sca_blocked`
  `msg_wait`

- **Arguments**

`nlimit`	input/output	integer
`mdir`	input	integer
`mp`	input	integer
`begin`	input	real(KIND=dp)
`final`	input	real(KIND=dp)
`shove`	input	real(KIND=dp)

- **Comments**
  Only exists in parallel version of DL_MESO_DPD.

import

- **Header records**
  `SUBROUTINE import (nlimit, mdir, mp, begin, final, shove)`

- **Function**
  Imports particle properties (forces, virials) from boundary halo for integration schemes that only require one set of forces to be calculated.

- **Dependencies**
  `error`
  `global_sca_and`
  `msg_receive_unblocked`
  `msg_receive_sca_blocked`
  `msg_send_blocked`
  `msg_send_sca_blocked`
  `msg_wait`

- **Arguments**

`nlimit`	input/output	integer
`mdir`	input	integer
`mp`	input	integer
`begin`	input	real(KIND=dp)
`final`	input	real(KIND=dp)
`shove`	input	real(KIND=dp)

- **Comments**
  Only exists in parallel version of DL_MESO_DPD. This routine is suitable for force integrators using the standard (MD) Velocity Verlet algorithm, including the Lowe-Andersen and Peters thermostats.

import_dpdvv

- **Header records**
  `SUBROUTINE import_dpdvv (nlimit, mdir, mp, begin, final, shove, first)`

- **Function**
  Imports particle properties (forces, virials) from boundary halo for integration schemes that require two separate sets of forces to be calculated.

- **Dependencies**
  `error`
  `global_sca_and`
  `msg_receive_unblocked`

msg_receive_sca_blocked
msg_send_blocked
msg_send_sca_blocked
msg_wait

- **Arguments**

nlimit	input/output	integer
mdir	input	integer
mp	input	integer
begin	input	real(KIND=dp)
final	input	real(KIND=dp)
shove	input	real(KIND=dp)
first	input	logical

- **Comments**
  Only exists in parallel version of DL_MESO_DPD. Setting first to .true. imports both sets of forces (constant and variable), while setting to .false. imports the variable force set and virials. This routine is suitable for integrators where types of forces need to be separated and/or recalculated, e.g. the DPD Velocity Verlet scheme and Stoyanov-Groot thermostat.

importdensity

- **Header records**
  SUBROUTINE importdensity (nlimit, mdir, mp, begin, final)

- **Function**
  Imports local densities from boundary halo.

- **Dependencies**
  error
  global_sca_and
  msg_receive_unblocked
  msg_receive_sca_blocked
  msg_send_blocked
  msg_send_sca_blocked
  msg_wait

- **Arguments**

nlimit	input/output	integer
mdir	input	integer
mp	input	integer
begin	input	real(KIND=dp)
final	input	real(KIND=dp)
shove	input	real(KIND=dp)

- **Comments**
  Only exists in parallel version of DL_MESO_DPD.

export

- **Header records**
  SUBROUTINE export (nlimit, mdir, mp, begin, final, shove) (Parallel version)
  SUBROUTINE export (nlimit, mdir, begin, final, shove) (Serial version)

- **Function**
  Exports particle data (positions, velocity) to neighbouring domain as boundary halo.

- **Dependencies**
  `error`
  `global_sca_and` (Parallel version only)
  `msg_receive_unblocked` (Parallel version only)
  `msg_receive_sca_blocked` (Parallel version only)
  `msg_send_blocked` (Parallel version only)
  `msg_send_sca_blocked` (Parallel version only)
  `msg_wait` (Parallel version only)

- **Arguments**

`nlimit`	input/output	integer
`mdir`	input	integer
`mp`	input	integer
`begin`	input	real(KIND=dp)
`final`	input	real(KIND=dp)
`shove`	input	real(KIND=dp)

exportvelocity

- **Header records**
  `SUBROUTINE exportvelocity (nlimit, mdir, mp, begin, final, shove)` (Parallel version)
  `SUBROUTINE exportvelocity (nlimit, mdir, begin, final)` (Serial version)

- **Function**
  Exports particle velocities to neighbouring domain as boundary halo for recalculation of dissipative forces (as required for DPD Velocity Verlet algorithm).

- **Dependencies**
  `error`
  `global_sca_and` (Parallel version only)
  `msg_receive_unblocked` (Parallel version only)
  `msg_receive_sca_blocked` (Parallel version only)
  `msg_send_blocked` (Parallel version only)
  `msg_send_sca_blocked` (Parallel version only)
  `msg_wait` (Parallel version only)

- **Arguments**

`nlimit`	input/output	integer
`mdir`	input	integer
`mp`	input	integer
`begin`	input	real(KIND=dp)
`final`	input	real(KIND=dp)
`shove`	input	real(KIND=dp)

exportdensity

- **Header records**
  `SUBROUTINE exportdensity (nlimit, mdir, mp, begin, final, shove)` (Parallel version)
  `SUBROUTINE exportdensity (nlimit, mdir, begin, final)` (Serial version)

- **Function**
  Exports particle data (local densities) to neighbouring domain as boundary halo for calculation of many-body DPD interaction forces.

- **Dependencies**
  `error`
  `global_sca_and` (Parallel version only)
  `msg_receive_unblocked` (Parallel version only)
  `msg_receive_sca_blocked` (Parallel version only)
  `msg_send_blocked` (Parallel version only)
  `msg_send_sca_blocked` (Parallel version only)
  `msg_wait` (Parallel version only)

- **Arguments**

`nlimit`	input/output	integer
`mdir`	input	integer
`mp`	input	integer
`begin`	input	real(KIND=dp)
`final`	input	real(KIND=dp)
`shove`	input	real(KIND=dp)

deportdata

- **Header records**
  `SUBROUTINE deportdata (nlimit)`

- **Function**
  Applies deport of particles from boundary halos to neighbouring domains and/or periodic boundary conditions.

- **Dependencies**
  `deport` (Parallel version only)

- **Arguments**

`nlimit`	input/output	integer

importdata

- **Header records**
  `SUBROUTINE importdata (nlimit)`

- **Function**
  Applies import of particle forces from boundary halos of neighbouring domains and/or across periodic boundary conditions.

- **Dependencies**
  `import` (Parallel version only)

- **Arguments**

`nlimit`	input/output	integer

- **Comments**
  This subroutine is applicable for integrators/thermostats using single sets of particle forces (i.e. standard Velocity Verlet, Lowe-Andersen, Peters).

importdata_dpdvv

- **Header records**
  `SUBROUTINE importdata_dpdvv (nlimit)`
- **Function**
  Applies import of particle forces (constant and variable) from boundary halos of neighbouring domains and/or across periodic boundary conditions.
- **Dependencies**
  `import_dpdvv` (Parallel version only)
- **Arguments**
  `nlimit` input/output integer
- **Comments**
  This subroutine is applicable for integrators/thermostats using two sets of particle forces (i.e. DPD Velocity Verlet, Stoyanov-Groot).

importforcedata

- **Header records**
  `SUBROUTINE importforcedata (nlimit)`
- **Function**
  Applies import of variable particle forces and virials from boundary halos of neighbouring domains and/or across periodic boundary conditions.
- **Dependencies**
  `import_dpdvv` (Parallel version only)
- **Arguments**
  `nlimit` input/output integer
- **Comments**
  This subroutine is used for importing recalculated forces for particles while using the DPD Velocity Verlet algorithm.

importdensitydata

- **Header records**
  `SUBROUTINE importdensitydata (nlimit)`
- **Function**
  Applies import of local densities for particles from boundary halos of neighbouring domains and/or across periodic boundary conditions.
- **Dependencies**
  `importdensity` (Parallel version only)
- **Arguments**
  `nlimit` input/output integer
- **Comments**
  This subroutine is essential for DPD calculations involving many-body (density dependent) interactions.

exportdata

- **Header records**
  `SUBROUTINE exportdata (nlimit)`
- **Function**
  Applies export of particle properties (positions, velocities) to boundary halos of neighbouring domains and/or across periodic boundary conditions.
- **Dependencies**
  `export`
- **Arguments**
  `nlimit` input/output integer
- **Comments**
  Non-periodic boundaries do not require boundary halos: these are excluded from the export of particle properties to prevent interactions between particles near opposite boundaries.

exportvelocitydata

- **Header records**
  `SUBROUTINE exportvelocitydata (nlimit)`
- **Function**
  Applies export of particle velocities to boundary halos of neighbouring domains and/or across periodic boundary conditions for recalculation of dissipative forces.
- **Dependencies**
  `exportvelocity`
- **Arguments**
  `nlimit` input/output integer
- **Comments**
  Non-periodic boundaries do not require boundary halos: these are excluded from the export of particle velocities.

exportdensitydata

- **Header records**
  `SUBROUTINE exportdensitydata (nlimit)`
- **Function**
  Applies export of local densities for particles to boundary halos of neighbouring domains and/or across periodic boundary conditions for calculation of many-body DPD interaction forces.
- **Dependencies**
  `exportdensity`
- **Arguments**
  `nlimit` input/output integer
- **Comments**
  Non-periodic boundaries do not require boundary halos: these are excluded from the export of particle local densities.

11.2.11 start_module

This module requires the modules `constants`, `variables`, `error_module`, `comms_module`, `numeric_container` and `parse_utils` to be loaded beforehand.

start

- **Header records**
  `SUBROUTINE start`
- **Function**
  Sets up starting configuration for DPD calculations.
- **Dependencies**
  `duni`
  `mtrnd`
  `revive`
  `boxfit`
  `readconfig`
  `initialvelocity`
  `error`
  `global_sca_and`
  `global_sum_dble`
- **Comments**
  If the user does not provide initial particle positions (with optional velocities and force) in a `CONFIG` file and the system is not restarted, all unbonded particles are assigned their species randomly according to the numbers required: this may be changed by the user.

revive

- **Header records**
  `SUBROUTINE revive (key)`
- **Function**
  Saves and reads restart configurations.
- **Dependencies**
  `export*` files (if reading restart configuration)
- **Arguments**

`key`	input	integer

readconfig

- **Header records**
  `SUBROUTINE readconfig`
- **Function**
  Reads initial system configuration (positions, velocities, forces) from DL_POLY-style `CONFIG` file.
- **Dependencies**
  `CONFIG` file
  `initialvelocity`

`global_sca_and`
`global_sum_sca_int`
`error`

- **Comments**
  The number of particles in CONFIG file must match up with the number described in the corresponding `CONTROL` file, as should the information in the `FIELD` and `MOLECULE` files for any bonded particles. The periodic boundary key `imcon`[52] is ignored, since all DL_MESO systems are orthorhombic.

initialvelocity

- **Header records**
  `SUBROUTINE initialvelocity`

- **Function**
  Initializes particle velocities randomly in system to give required system temperature.

- **Dependencies**
  `duni`
  `global_sum_dble`

boxfit

- **Header records**
  `SUBROUTINE boxfit (side, idd, ds, displ, npls)`

- **Function**
  Sets up lattice for unbonded particles in processor domain.

- **Dependencies**
  None

- **Arguments**

`side`	input	real(KIND=dp)
`idd`	input	integer
`ds`	input	real(KIND=dp)
`displ`	output	real(KIND=dp)
`npls`	output	integer

11.2.12 config_module

This module requires the modules `constants`, `variables`, `comms_module`, `numeric_container`, `parse_utils`, `domain_module` and `surface_module` to be loaded beforehand.

sysdef

- **Header records**
  `SUBROUTINE sysdef`

- **Function**
  Reads in system data, determines simulation properties and prints to `OUTPUT` file.

- **Dependencies**
  `datsys`
  `bondsys`
  `domain`
  `surfacenodes`
  `forgen`
  `elecgen`
  `error`
  `global_sca_and`
- **Arguments**
  `idnode` input integer

datsys

- **Header records**
  `SUBROUTINE datsys`
- **Function**
  Reads in system data from `CONTROL` file.
- **Dependencies**
  `CONTROL` file
  `getdble`
  `getint`
  `getword`
  `error`
  `global_sca_and`

bondsys

- **Header records**
  `SUBROUTINE bondsys`
- **Function**
  Reads in molecule and bond data from `MOLECULE` and `FIELD` files.
- **Dependencies**
  `MOLECULE` file
  `FIELD` file
  `getdble`
  `getint`
  `getword`
  `error`
  `global_sca_and`

zero

- **Header records**
  `SUBROUTINE zero`
- **Function**
  Sets step counters, initial time parameters, system parameters, accumulators for statistical properties and long-range potential corrections to zero, initializes random number generators.

- **Dependencies**
  `duni`
  `mtrnd`

forgen

- **Header records**
  `SUBROUTINE forgen`
- **Function**
  Sets up arrays for potential and force parameters.
- **Dependencies**
  `error`
  `global_sca_and`

elecgen

- **Header records**
  `SUBROUTINE elecgen`
- **Function**
  Sets up electrostatic parameters for self-interaction and charged system corrections.
- **Dependencies**
  `error`
  `global_sca_and`

11.2.13 field_module

This module requires the modules `constants`, `variables`, `bond_module`, `manybody_module`, `ewald_module`, `surface_module`, `numeric_container` and `domain_module` to be loaded beforehand.

Different versions of each subroutine are available in this module for different integrators and/or thermostats:

- DPD thermostat with standard (molecular dynamics) Velocity Verlet integration (MD-VV)[54] (`mdvv`)
- DPD thermostat with DPD Velocity Verlet integration (DPD-VV)[3] (`dpdvv`)
- Lowe-Andersen thermostat[27] (`lowe`)
- Peters thermostat[38] (`peters`)
- Stoyanov-Groot thermostat[49] (`stoyanov`)

forces_*

- **Header records**
  `SUBROUTINE forces_* (nlimit)`
- **Function**
  Calculates all forces between particles, particularly pairwise forces within cut-off radius.

- **Dependencies**
  `mtrnd` (`duni`, `gaussmp`)
  `manybody_force`
  `manybody_potential`
  `ewald_real`
  `ewald_reciprocal`
  `bondforcesglobal`
  `bondforceslocal`
  `hardreflect`

- **Arguments**

`nlimit`	input	integer

- **Comments**
  The random force is calculated using a uniform random number generator[13] (by default the Mersenne twister generator `mtrnd`), i.e.

$$\zeta_{ij} \approx \sqrt{12}\,(u - 0.5)\,,$$

  which produces statistically similar results[8] and is computationally more efficient than using the Gaussian random number subroutine `gaussmp`, although this or the `duni` random number generator may be substituted if required.

dragforces_dpdvv

- **Header records**
  `SUBROUTINE dragforces_dpdvv (nlimit)`

- **Function**
  Recalculates dissipative forces between particles within cut-off radius.

- **Dependencies**
  None.

- **Arguments**

`nlimit`	input	integer
`idnode`	input	integer

- **Comments**
  Only required for DPD Velocity Verlet (DPD-VV) integration.

plcfor_*

- **Header records**
  `SUBROUTINE plcfor_*`

- **Function**
  Sets up parallel link-cells structure and calculates forces (including those involving particles in boundary halo).

- **Dependencies**
  `exportdata`
  `parlnk`
  `local_density`
  `importdensitydata`
  `exportdensitydata`

`forces_*`
`importdata` (`importdata_dpdvv`)

11.2.14 integrate_module

This module requires preloading of the `constants`, `variables`, `domain_module`, `surface_module`, `comms_module`, `error_module`, `numeric_container` and `field_module` modules. Like the `field_module`, different versions of the subroutine (`mdvv`, `dpdvv`, `lowe`, `peters`, `stoyanov`) are available for each thermostat/integrator.

verlet_*

- **Header records**
  `SUBROUTINE verlet_* (stage)`
- **Function**
  Solves the equations of motion using the Velocity Verlet scheme[54].
- **Dependencies**
  `hardreflect`
  `deportdata`
  `error`
  `exportvelocitydata` (DPD-VV only)
  `dragforces_dpdvv` (DPD-VV only)
  `importforcedata` (DPD-VV only)
  `global_sca_sca_int`
  `global_sca_sca_dble`
  `global_sum_dble` (Lowe, Peters, Stoyanov)
  `global_sum_int` (Lowe, Peters, Stoyanov)
- **Arguments**
  `stage` input integer
- **Comments**
  All versions of this subroutine use the standard Velocity Verlet algorithm to integrate the equations

$$\frac{d\vec{v}_i}{dt} = \frac{\vec{F}_i}{m_i}$$

$$\frac{d\vec{r}_i}{dt} = \vec{v}_i$$

  Both the MD-VV and DPD-VV algorithms use dissipative and random forces as the thermostat, with the DPD-VV algorithm repeating the calculation of dissipative forces at the end of the time step. The Lowe-Andersen, Peters and Stoyanov-Groot thermostats are applied after all other forces are integrated.

verlet_*_lang

- **Header records**
  `SUBROUTINE verlet_*_lang (stage)`
- **Function**
  Solves the equations of motion using the Velocity Verlet scheme coupled with a Langevin barostat[22].
- **Dependencies**
  `hardreflect`

deportdata
error
exportvelocitydata (DPD-VV only)
dragforces_dpdvv (DPD-VV only)
importforcedata (DPD-VV only)
global_sca_sca_int
global_sca_sca_dble
global_sum_dble
global_sum_int (Lowe, Peters, Stoyanov)

- **Arguments**
  stage input integer
- **Comments**
  The Langevin barostat is configured to apply the following piston force (rate of change of momentum) in dimension α

$$\dot{p}_{g,\alpha} = V(P_\alpha - P_0) + \frac{1}{N_f}\sum_i m_i v_i^2 - \gamma_p p_{g,\alpha} + \sigma_p \zeta_{p,\alpha}$$

  with the option of keeping the system isotropic by using equal values of P_α and $\zeta_{p,\alpha}$ for all dimensions. (This may be switched off if imposing constant surface tensions at given planar surfaces.) Rescaling of the simulation volume takes place in the first stage coupled with the integration of particle forces, while iteration of the barostat force to achieve convergence of particle velocities (to machine precision) takes place in the second stage.

verlet_*_berend

- **Header records**
  SUBROUTINE verlet_*_berend (stage)
- **Function**
  Solves the equations of motion using the Velocity Verlet scheme coupled with the Berendsen barostat[2].
- **Dependencies**
  hardreflect
  deportdata
  error
  exportvelocitydata (DPD-VV only)
  dragforces_dpdvv (DPD-VV only)
  importforcedata (DPD-VV only)
  global_sca_sca_int
  global_sum_dble
  global_sum_int (Lowe, Peters, Stoyanov)
- **Arguments**
  stage input integer
- **Comments**
  The Berendsen barostat rescales particle positions and system sizes in dimension α by the factor

$$\mu_\alpha = 1 + \frac{\beta}{\tau_p}\Delta t(P_\alpha - P_0)$$

  with the option of keeping the system isotropic by using equal values of P_α for all dimensions. (This may be switched off if imposing constant surface tensions at given planar surfaces.) Rescaling of the simulation volume takes place in the first stage, while calculation of the rescaling factor μ_α takes place in the second stage after force integration.

11.2.15 statistics_module

This module requires the modules `constants`, `variables`, `start_module`, `comms_module` and `numeric_container` to be loaded beforehand.

printout

- **Header records**
  `SUBROUTINE printout (time, lbegin)`
- **Function**
  Writes summary of simulation at the current time step to `OUTPUT` file.
- **Dependencies**
  None
- **Arguments**

`time`	input	real(KIND=dp)
`lbegin`	input	logical

- **Comments**
  The logical `lbegin` indicates that the column titles are to be printed before the data.

corout

- **Header records**
  `SUBROUTINE corout (time)`
- **Function**
  Writes statistical data to `CORREL` data file.
- **Dependencies**
  None
- **Arguments**

`time`	input	real(KIND=dp)

histout

- **Header records**
  `SUBROUTINE histout (time)`
- **Function**
  Writes trajectory data (including virial) to `HISTORY*` data files.
- **Dependencies**
  None
- **Arguments**

`time`	input	real(KIND=dp)

result

- **Header records**
  `SUBROUTINE result`
- **Function**
  Writes final summary of simulation to `OUTPUT` file.
- **Dependencies**
  `revive`
- **Arguments**

`idnode`	input	integer
`nodes`	input	integer

statis

- **Header records**
  `SUBROUTINE statis`
- **Function**
  Calculates statistical properties of system, including rolling averages and fluctuations.
- **Dependencies**
  `sclsum`
  `global_sum_dble`

11.3 DL_MESO_DPD Data Files

11.3.1 Input files

All input files for DL_MESO_DPD must be in ANSI text format, with keywords (where necessary) and numerical values separated from each other with spaces or commas: tabs are currently *not* recognised by the parsing utilities.

Define system: `CONTROL`

The `CONTROL` file can be obtained either via use of the DL_MESO GUI or by editing existing files of that name, such as those in the `DEMO/DPD` directory. With the exception of the simulation name on the first line, its format consists of keywords for system parameters (of which the first four characters are recognised by DL_MESO_DPD) followed by their values with spaces separating them. The majority of keywords are followed by a single value: these are summarized in Table 11.1.

While most parameters are defined by a single value, certain keywords are followed by more than one parameter:

- essential data (name, number, mass, interaction parameters, constant external forces)[4] for individual species

  `SPEC` ***name*** n_i m_i A_i B_i C_i γ_i/Γ_i $F_{x,i}$ $F_{y,i}$ $F_{z,i}$

[1] If this is specified, the system is assumed to be cubic.
[2] Takes the largest out of the maximum bond length, interaction cutoff `rcut` or electrostatic cutoff `relec`.
[3] Numbers of bonded particles come from the `MOLECULE` file and therefore do not need to be included in `CONTROL`.
[4] Parameters after the interaction length B_i are not essential for basic DPD calculations, although C_i is additionally required for charged systems; all omitted parameters will be set to zero with the exception of drag coefficients or frequencies, which will be set to the system-wide value.

Table 11.1: Keywords for CONTROL file with single parameters

keyword	meaning	valid values	default value
VOL	(initial) volume of system[1]	real number > 0	-
TEMP	specified system temperature	real number > 0	-
PRESS	specified system pressure	real number > 0	-
TSTEP	duration of calculation timestep	real number > 0	-
RCUT	interaction cutoff radius	real number > 0	-
RMBCUT	many-body interaction cutoff radius	real number > 0	rcut
RHALO	size of boundary halo	real number > 0	Largest required[2]
RELEC	short-range electrostatic cutoff radius	real number > 0	rcut
SRFZCUT	surface interaction cutoff distance	real number > 0	rcut
GAMMA	system-wide drag coefficient/collision frequency	real number ≥ 0	-
TIMJOB	maximum time available to run calculation	real number > 0	-
TCLOSE	time required to shut down calculation	real number > 0	-
KRES	calculation restart parameter	integer (0-3)	-
KTYPE	potential energy function selection	integer (0-3)	-
NSYST	total number of *unbonded* particles[3]	integer > 0	-
NRUN	number of calculation timesteps	integer > 0	-
NSEQL	number of equilibration timesteps	integer ≥ 0	-
NSBTS	interval for equilibration temperature scaling	integer ≥ 0	-
NSTK	size of statistical data stack	integer > 0	-
NSBPO	interval for printing data to OUTPUT file	integer ≥ 0	-
NSBSV	interval for saving trajectory data to HISTORY file(s)	integer ≥ 0	-
ISCORR	interval for saving statistical data to CORREL file	integer ≥ 0	-
NSPE	number of particle species	integer > 0	-
NMOLDEF	number of molecule types	integer ≥ 0	-
LCORR	switch for saving statistical data to CORREL file	Boolean	.false.
LSBSV	switch for saving trajectory data to HISTORY file	Boolean	.false.
LTEMP	switch for temperature scaling during equilibration	Boolean	.false.
LISOPRS	switch for isotropic pressure tensors in barostat	Boolean	.true.
LBOND	switch for modelling bonds between particles	Boolean	.false.
LANGLE	switch for modelling bond angles	Boolean	.false.
LDIHED	switch for modelling bond dihedrals	Boolean	.false.
LGBND	switch for globally storing bond data	Boolean	.false.

- name and number of individual molecules

$$\texttt{MOLE}\ \textit{name}\ \ n_i$$

- unlike conservative potential parameters, interaction lengths and drag coefficients/collision frequencies[5]

$$\texttt{UNLIKE}\ \ i\ \ j\ \ A_{ij}\ \ B_{ij}\ \ \gamma_{ij}/\Gamma_{ij}$$

- many-body DPD parameters for each species and interacting species pair

$$\texttt{MANY}\ \ i\ \ j\ \ A_{ij}\ \ B_{ij}\ \ C_{ij}\ \ D_{ij}\ \ E_{ij}$$

- the dimensions for non-cubic orthorhombic systems

$$\texttt{DIM}\ \ L_x\ \ L_y\ \ L_z$$

- the thermostat/integrator type and its parameters (see Table 10.1)

$$\texttt{THERM itype}\ \ a\ \ b\ \ c\ \ d$$

[5]Setting any of these parameters to 0.0 will make DL_MESO_DPD use the default values, i.e. $A_{ij} = \sqrt{A_i A_j}$, $B_{ij} = \frac{1}{2}(B_i + B_j)$ and γ_{ij} or Γ_{ij} takes the system-wide value.

- the barostat type and its parameters (see Table 10.2)

  `BARO btype` a b c d

- the electrostatics method and its parameters (see Table 10.3)

  `ELEC etype` a b c d

- maximum number of periodic images in each dimension for Ewald sums[6]

  `KMAX` k_1^{max} k_2^{max} k_3^{max}

- non-periodic surface type, applicable boundaries and (where applicable) surface parameters

  `SURFACE srftype srfx srfy srfz` etc.

- constant external (body) forces per particle for *every* species

  `BODY` F_x F_y F_z

Note that not all parameters have to be included in the `CONTROL` file if they are not required for the given type of bond, thermostat, barostat, electrostatics, surface etc. Superfluous parameters for particular systems (e.g. specified pressure for constant volume simulations) can be omitted from the `CONTROL` file without causing runtime problems. If the user wishes to include new parameters and/or keywords in the `CONTROL` file, modifications to the parameter recognition loop in the `datsys` subroutine (`config_module`) will be required.

Define molecules: `MOLECULE` and `FIELD`

A single `MOLECULE` file and a single `FIELD` file are required for including molecules in DPD calculations. The `MOLECULE` file includes the species and relative Cartesian coordinates for the particles (with the origin at the centre of the cube) in all molecule types (up to eight by default); each molecule type is formatted thus:

`MOLECULE` n ***cubesize***

s_1 x_1 y_1 z_1

s_2 x_2 y_2 z_2

...

s_n x_n y_n z_n

where n is the number of particles in the molecule (default maximum, `mxbonds`, of 500), ***cubesize*** is the size of cube within which the molecule lies and used to insert it into the system (which can be no bigger than the smallest dimension for the entire system), while the species number and Cartesian coordinates (x, y, z) of particle i are given by s_i, x_i, y_i and z_i respectively. If the user is likely to change the order or number of species in the `CONTROL` file, the names of species as defined in the `CONTROL` file can be used in place of s_i.

The `FIELD` file contains both the properties of the bonds, angles and dihedrals required and the bond pairs, bond angle triples and bond dihedral quadruples for each molecule type. The bond/angle/dihedral properties are listed first in the following formats:

`BOND` ***bondtype*** a b c d

`ANGLE` ***angletype*** a b c d

`DIHEDRAL` ***dihedraltype*** a b c d

[6] If any of the values are omitted, they are given the values of k_1^{max}.

where the bond/angle/dihedral types are integers as listed in Section 10.6, and a, b, c and d are the corresponding parameters (double precision floating point numbers) as per Tables 10.4, 10.5 and 10.6.

The relationships between particles for each molecule type follow, which are preceded by the line indicating the numbers of bond pairs, angle triples and dihedral quadruples to follow (if no angles or dihedrals are to be listed, these values can be omitted):

`MOLECULE` n_{bond} n_{angle} $n_{dihedral}$

Bond pairs then follow in the format:

$$\begin{array}{l} m_{1,1}\ m_{1,2}\ b_1 \\ m_{2,1}\ m_{2,2}\ b_2 \\ \ldots \\ m_{n,1}\ m_{n,2}\ b_n \end{array}$$

where $m_{i,j}$ is the particle number of the molecule for the ith bond pair and the jth particle in the pair, and b_i is an integer indicating the bond as previously listed (e.g. 2 uses the second set of listed bond properties) for pair i.

Similarly, bond angle triples are formatted thus:

$$\begin{array}{l} m_{1,1}\ m_{1,2}\ m_{1,3}\ a_1 \\ m_{2,1}\ m_{2,2}\ m_{2,3}\ a_2 \\ \ldots \\ m_{n,1}\ m_{n,2}\ m_{n,3}\ a_n \end{array}$$

while bond dihedral quadruples are formatted

$$\begin{array}{l} m_{1,1}\ m_{1,2}\ m_{1,3}\ m_{1,4}\ d_1 \\ m_{2,1}\ m_{2,2}\ m_{2,3}\ m_{2,4}\ d_2 \\ \ldots \\ m_{n,1}\ m_{n,2}\ m_{n,3}\ m_{n,4}\ d_n \end{array}$$

with a_i and d_i indicating the angle and dihedral properties to use for the triple/quadruple.

Both of these files can be created using the supplied C++ program `molecule-generate` in the directory `DPD/utility` – see Appendix B for more details – although some editing is currently required to specify the bond/angle/dihedral types and properties. Example files in the `DEMO/DPD` directory can be examined for this purpose.

Initial state: `CONFIG`

An optional `CONFIG` file can be included to define the initial state of the system, which can include the positions, velocities and forces for each particle[7]. Before this information is included, five lines of information (of which two are mandatory) have to be included:

- The simulation name (80 characters)
- The `CONFIG` file key `levcfg` (integer), the periodic boundary key `imcon` (integer), the number of particles in the file (integer, optional) and the configuration energy (real, optional)
- The x-, y- and z-components for the x-axis vector (real, optional)

[7]This file is formatted identically to `CONFIG` files used in DL_POLY[48, 52], except that the origin is set as the back bottom left corner of the simulation volume instead of the centre.

- The x-, y- and z-components for the y-axis vector (real, optional)
- The x-, y- and z-components for the z-axis vector (real, optional)

The file key `levcfg` is set depending on the information available for each particle: 0 for positions only, 1 for positions and velocities or 2 for positions, velocities and forces. If particle velocities are not specified, these are generated at random to produce a distribution corresponding to the required system temperature, while unknown forces are set to zero. The simulation name, number of particles in the file and configuration energy are not read by DL_MESO_DPD and can thus be ignored (although the line for the simulation name must remain). The axes vectors are not used by DL_MESO_DPD and the lines specifying them are skipped over if the value of `imcon` is set to a value greater than zero: if set to zero, the lines can be omitted.

Each particle is represented by a block record, with at least two lines of information:

- The species name (8 characters) or number (integer) and the global particle number (integer, optional)
- The x-, y- and z-coordinates for the particle (real)
- The x-, y- and z-components of particle velocity (real, if `levcfg`> 0)
- The x-, y- and z-components of force on the particle (real, if `levcfg`> 1)

If global particle numbers are not included, these are generated automatically for the particles in the order specified by the `CONFIG` file. Care should be taken that any particles belonging to molecules are numbered correctly, since the bond information is assigned in an identical fashion to unspecified systems, i.e. numbering after all loose particles in the relative order specified by the `MOLECULE` and `FIELD` files.

`CONFIG` files can be created from restart files of previous simulations using the supplied Fortran90 program `exportconfig` in the directory `DPD/utility`; see Appendix B for more details.

11.3.2 Output files

General output file: `OUTPUT`

This ANSI text file is generated by all DPD calculations and contains:

- The system and bond/angle/dihedral properties used for calculations.
- Domain decomposition details (Parallel version only).
- The starting positions and velocities of a particle sample.
- The calculation time, current values and rolling averages for the total energy, potential energy (total, electrostatic, and from bond stretching, angles and dihedrals), virial, kinetic energy, pressure and temperature every `NSBPO` time steps.
- Final averages and fluctuations (standard deviations) over all time steps after equilibration.
- The final positions and velocities of a particle sample.
- Elapsed and average times for the calculation.

Restart file(s): `export*`

Each processing unit produces a restart file with a name beginning with `export` every `ndump` time steps. This binary file contains the following information for the time step:

- Name of DPD calculation.
- Numbers of particles, bonds, angles and dihedrals in the processing unit.
- Specified temperature, number of time steps, system volume.
- Current state of random number generators.
- Particle global identity numbers, species and molecule numbers
- Bond, angle and dihedral tables.
- Particle Cartesian coordinates, velocities, forces and virials.
- Current time step and statistical properties (current values, cumulative sums and fluctuations, rolling averages and stacks).

In combination, the `export*` files can restore a stopped DPD calculation. They can also be used to generate a plot at the given time step using the utility `exportimage` in the `DPD/utility` directory; Appendix B gives instructions for its use.

Trajectory file(s): `HISTORY*`

If the parameter `lsbsv` is set to true, each processing unit will generate a trajectory file with a name beginning with `HISTORY` every `nsbsv` time steps. Each binary file contains some of the system properties — including an identity number for each molecule — followed by the positions, velocities and virial for each particle in the domain.

The `HISTORY*` files can be used with the utility `traject` in the `DPD/utility` directory to produce plottable VMD files, with sets of bonded particles represented as residues. The utility `local` in the same directory can calculate localized properties (e.g. temperature, composition, pressure) from the same files and produces VTK files with these properties as cell data. Appendix B gives instructions for their use.

Statistical data file: `CORREL`

If the parameter `lcorr` is set to true, DL_MESO_DPD will generate an ANSI text file containing statistical data every `iscorr` time steps, which can later be imported into a spreadsheet or used by graph-plotting software. The data is formatted thus:

$$t \;\; E_{tot} \;\; E_{pot,tot} \;\; E_{pot,elec} \;\; E_{pot,bond} \;\; E_{pot,angle} \;\; E_{pot,dihed} \;\; P \;\; V \;\; T \;\; \langle r_{bond} \rangle \;\; \langle \theta_{angle} \rangle \;\; \langle \phi_{dihed} \rangle$$

where t is the time, E energy (*tot* denoting total, *pot* potential), P pressure, V volume, T temperature, $\langle r_{bond} \rangle$ the mean bond length, $\langle \theta_{angle} \rangle$ the mean bond angle and $\langle \phi_{dihed} \rangle$ the mean bond dihedral. The mean bond angle and dihedral are expressed in degrees.

Chapter 12

DL_MESO DPD Examples

Test cases for Dissipative Particle Dynamics simulations using DL_MESO – including the required input and sample output files – can be found in the `DEMO/DPD` subdirectory. These can be divided into problems that should, by default, be run using the serial[1] and parallel versions of DL_MESO_DPD; the latter have been tested with 96 processing units but should also run on smaller and larger numbers. They consist of the following cases.

12.1 Mixture_Ser

This *serial* simulation example consists of 1000 particles with 3 species. Species 1 has a population of 333, species 2 has a population of 333 and species 3 has a population of 334 particles. All particle types have identical sizes and masses but different energy parameters, using the default values for unlike particle parameters. Figure 12.1 gives a snapshot of the system at the end of the simulation, demonstrating mixing between the three particle types (represented by different colours).

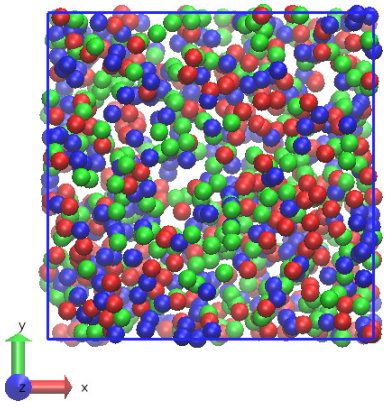
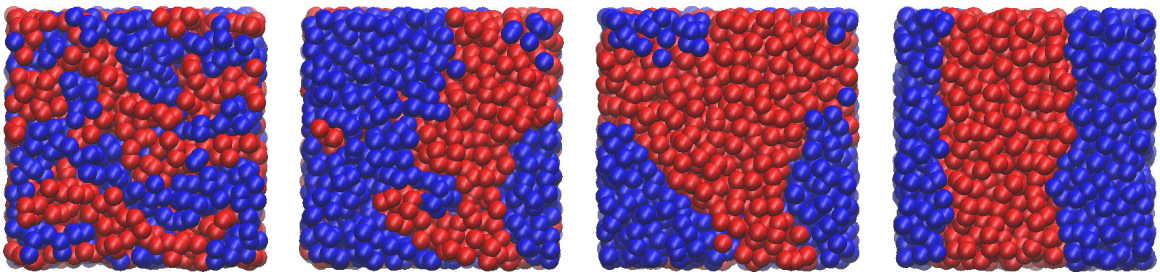


Figure 12.1: Visualization of system at final time step from DPD `Mixture_Ser` test case

12.2 PhaseSeparation

This *serial* simulation example consists of 3000 particles with 2 species. Species 1 has a population of 1500 and species 2 has a population of 1500 particles. Both particle types have identical sizes, masses and like-like energy parameters, but the unlike energy parameter has been set to a larger value to produce phase separation, which can clearly be seen in Figure 12.2. An `.AVI` video file of the first half of the simulation can be found in the subdirectory.

[1] These can also be run using the parallel version of DL_MESO_DPD, but care should be taken not to use too many processes due to either excessive communications or inability to insert molecules into the system.



(a) $t = 1$ (b) $t = 10$ (c) $t = 25$ (d) $t = 50$

Figure 12.2: Visualizations of DPD `PhaseSeparation` test case (red for particle type 1, blue for type 2)

12.3 Aggregate

This *serial* simulation example consists of 3000 unbonded particles and 30 molecules of 10 particles each with harmonic bonds between them. The unbonded particles and molecules are made up of different species with a higher energy parameter for unlike particle interactions. This causes the molecules to aggregate, which can be seen in Figure 12.3, a snapshot of the simulation.

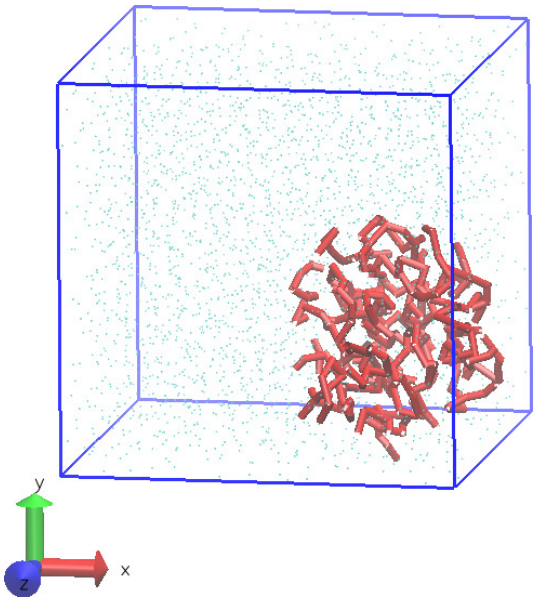
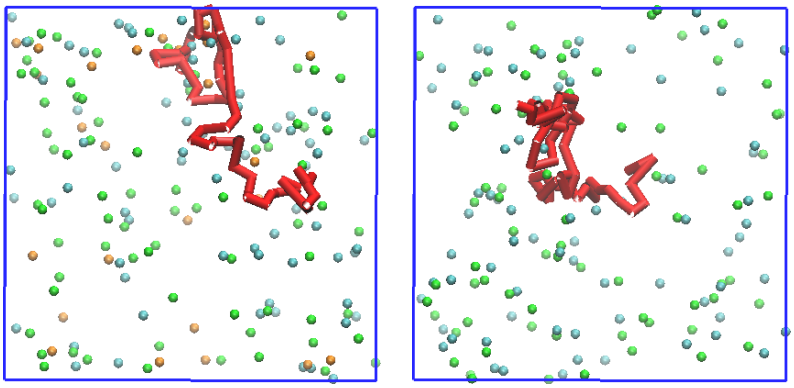


Figure 12.3: Visualization of system at $t = 2010$ from DPD `Aggregate` test case

12.4 Polyelectrolyte

This *serial* simulation example consists of a slightly hydrophobic polyelectrolyte molecule of 50 particles, each with a relative charge of +0.5, immersed in a salt solution of concentration $0.14M$[11]. The salt solution consists of 9900 neutral water particles, 75 cationic salt particles (net charge +1), 75 anions of charge −1 and 25 counterions of charge −1 to keep the system neutral. A similar simulation is included with the polyelectrolyte replaced with a neutral polymer of the same number of particles and the counterions replaced with water (`CONTROL-neutral`: this should be renamed to `CONTROL` to run the simulation and used with the same `FIELD` and `MOLECULE` files). Figure 12.4 gives a comparison between the polyelectrolyte and neutral polymer at the final time step, which have measured radii of gyration of 4.5 and 2.7 respectively.



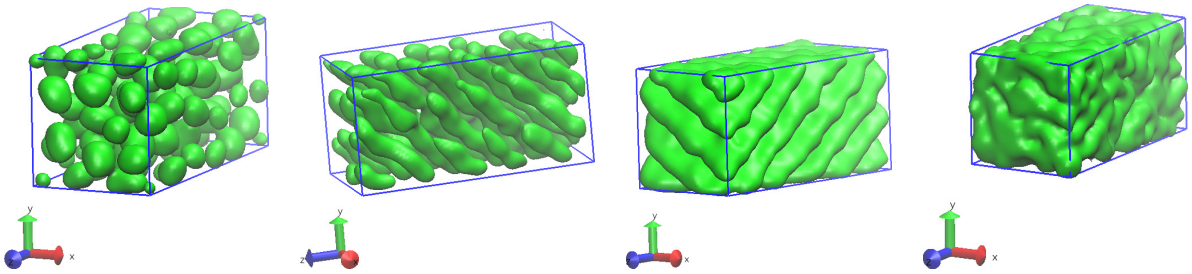
(a) Polyelectrolyte (b) Neutral polymer

Figure 12.4: Visualizations of DPD `Polyelectrolyte` test case: red for polyelectrolyte/polymer, green for salt cations, cyan for anions, orange for counterions (water omitted for clarity)

12.5 AmphiphileMesophases

This *serial* simulation example consists of four separate simulations, each with 12 000 particles consisting of dimers (molecules consisting of two particles, one hydrophilic and the other hydrophobic, with harmonic bonds of equilibrium length 0.5 between them) and unbonded monomers[24]. Defining the composition ϕ as the ratio of DPD particles within dimers to the total number of particles in the system, simulations with dimer compositions of 30%, 55%, 70% and 90% are provided with filenames `CONTROL-30`, `CONTROL-55`, `CONTROL-70` and `CONTROL-90` respectively. (Each of these files should be renamed to `CONTROL` to run the simulation, while the `FIELD` and `MOLECULE` files defining the amphiphile dimer molecule should be used for all four simulations.)

These systems provide four points on a phase diagram corresponding to isotropic dimer, hexagonal, lamellar and isotropic monomer phases respectively. The final configurations obtained for each phase can be seen in Figure 12.5, shown as isosurfaces of the hydrophobic particles to highlight the distinctions between the phases.



(a) $\phi = 0.30$ (b) $\phi = 0.55$ (c) $\phi = 0.70$ (d) $\phi = 0.90$

Figure 12.5: Visualizations of DPD `AmphiphileMesophases` test case at final time step (isosurfaces of hydrophobic particles)

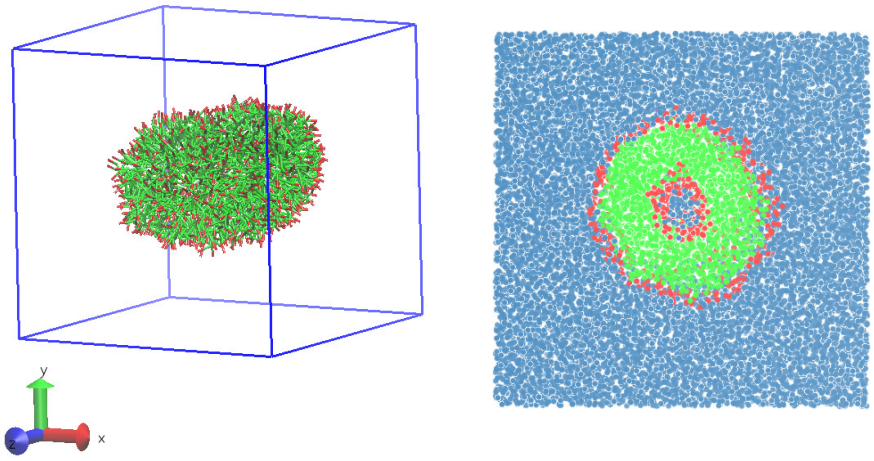
12.6 Mixture_Par

This *parallel* simulation example consists of 512 000 particles with 2 species. Species 1 has a population of 256 000 and species 1 has a population of 256 000 particles. The particle types have identical sizes and masses but different energy parameters, using the default values for unlike particle parameters.

12.7 VesicleFormation

This *parallel* simulation example consists of 37 440 unbonded water particles and 1008 molecules, each consisting of one hydrophilic head particle and three hydrophobic tail particles bonded together with stiff harmonic bonds

of equilibrium length 1.0 between them[58]. The molecules represent amphiphiles and during the course of the simulation self-assemble into a vesicle and encapsulate a number of water particles. Figure 12.6 shows the self-assembled vesicle, both in three dimensions and in a cross-section to show the encapsulated water. An `.AVI` video file of the simulation can be found in the subdirectory.



(a) Vesicle (water omitted for clarity) (b) Cross-section orthogonal to x-axis

Figure 12.6: Visualizations of DPD `VesicleFormation` test case at $t = 43000$ (red for hydrophile, green for hydrophobe, blue for water)

Appendix A

Manual compiliation and running of DL_MESO

A.1 DL_MESO_LBE

DL_MESO_LBE has been written in C++ in a modular form, where the main codes for serial and parallel running, `slbe.cpp`[1] and `plbe.cpp`, automatically include all the required modules. To compile the code and produce an executable in the working directory, at a command line either type:

- `c++ ../LBE/slbe.cpp -o lbe.exe` to compile the serial version[2], or
- `c++ ../LBE/plbe.cpp -o lbe.exe` to compile the parallel version

assuming that `c++` is the name of the available C++ compiler and `lbe.exe` is the name of the executable required.

After creating or copying the required input files (`lbin.sys` and `lbin.spa`) into the working directory, DL_MESO_LBE can then either be run at the command line (`lbe.exe` or `./lbe.exe` depending on the operating system) or launched using a run script.

By default, DL_MESO_LBE models compressible fluids using the Bhatnagar-Gross-Krook single relaxation time scheme and applies a correction of $\frac{\tau_f \vec{F}}{\rho}$ to the velocity for interfacial and body forces. To model incompressible fluids, calls to the following routines in `slbe.cpp` or `plbe.cpp` should have `Incom` appended to the end:

- `fInitializeSystem`
- `fReadInitialState`
- `fOutput*` or `fsOutput*`
- `fCollision*`
- `fPostCollBoundary`
- `fPostPropBoundary`

The Guo forcing scheme can be applied by appending `Guo` to the call for the collision scheme, e.g. `fCollisionBGKGuo`. To use multiple-relaxation-time (MRT) schemes, the call to `fCollisionBGK*` should be changed to `fCollisionMRT*` and the tuneable parameters can be modified in the `lbpMODEL.cpp` file for the required lattice model (D2Q9, D3Q15 or D3Q19).

Additional boundary conditions should be added to the `fPostCollBoundary*` or `fPostPropBoundary*` routines, depending on whether they are applied after collisions (and before propagation) or after propagation respectively.

[1] If preferred, an alternative version that uses a boundary layer, `slbecombine.cpp`, is available.
[2] `/` may need to be replaced by \ on computers running Windows.

Calls to alternative and/or additional output routines can be included in the main program files, provided they are called before the output file number (`qVersion`) is increased. If no fluid-fluid interaction forces are required, the call to `fInteractionForceShanChen` can be commented out to increase the calculational efficiency.

A.2 DL_MESO_DPD

The Fortran90 modules for DL_MESO_DPD must be compiled in a particular order to satisfy dependencies of shared variables and arrays:

- `constants`
- `variables`
- `numeric_container`
- `comms_module`
- `error_module`
- `parse_utils`
- `bond_module`
- `surface_module`
- `ewald_module`
- `manybody_module`
- `domain_module`
- `start_module`
- `config_module`
- `field_module`
- `integrate_module`
- `statistics_module`
- `run_module`
- `dlmesodpd`

If running DL_MESO_DPD in serial, the modules `comms_module` and `domain_module` should be replaced by `comms_module_ser` and `domain_module_ser` respectively.

To simplify the process, a makefile may be created either in the `DPD` directory or in the working directory to automatically compile the modules and build the executable. Examples of these for running in the `DPD` directory may be found in the `DPD/makefiles` directory and modified by the user: the compiler (after `FC=`) and flags (after `FFLAGS=`) may need changing depending on the Fortran90 compiler available. If invoking from the working directory, the modules for DL_MESO_DPD should either be preceded by the path, i.e. `../DPD/`, in the list of compile sources or the directive `VPATH=../DPD/` can be used before the source list; the latter strategy is used by the DL_MESO GUI when creating makefiles.

DL_MESO_DPD can be compiled by the command `make` if the makefile is called `Makefile`, or if it has a custom name (e.g. `Makefile-custom`) by the command:

- `make -f Makefile-custom`

The example makefiles will produce an executable with the name `dpd.exe`, which can be copied to the working directory (if necessary) and run at the command line after the required input files (`CONTROL` and, where applicable, `MOLECULE` and `FIELD`) are created or copied into the same directory. A `CONFIG` file can also be placed in the same directory if an initial configuration is required for a new simulation; `export*` files from a previous run can be used for restarting a previous simulation or providing a configuration for a new simulation, providing the number of processing units remains the same.

Prior to compliation, modifications may be made in the `constants.f90` file to the following maximum numbers:

- particles per process (`maxdim`)
- species (`mxspe`)
- molecule definitions (`mxmoldef`)
- bond interaction definitions (`mxbonddef`)
- particles per molecule (`mxmolsize`)
- bonds per molecule (`mxbonds`)
- size of statistical stack arrays (`mxstak`)

The maximum sizes of transfer buffers `maxbuf` and potentials (`mxpot`) are automatically set by the values of `maxdim` and `mxspe` respectively and do not need to be modified by the user. The value of `maxdim` should additionally take into account the number of particles likely to be included in boundary halos.

If using alternative many-body DPD interactions to the vapour-liquid example provided, the routines `manybody_force` and `manybody_potential` in `manybody_module.f90` should be modified by the user as necessary: the routine `local_density` should not be altered by the user but if an alternative weight function for calculating local densities is required, `weight_rho` can be changed.

Appendix B

DL_MESO Utilities

DL_MESO includes a number of utility programs which are not directly needed for Lattice Boltzmann or DPD simulations but are useful both for producing files required as inputs for those calculations and to process output files for visualization. These may be found in the `/LBE/utility` and `/DPD/utility` directories.

Compilation can either be carried out individually or collectively using makefiles: each utility directory includes a makefile to compile all the utilities therein and the working directory `/WORK` includes one to compile both sets for use with the GUI[1]. The latter can be invoked using the command

- `make -f Makefile-utils`

Some further details on these utilities can be found in the `README` files in the source directories.

B.1 DL_MESO_LBE

lbeplot3dgather

`lbeplot3dgather` is a utility written in C++ to gather Plot3D output files produced by the parallel version of DL_MESO_LBE and produce a single structure file (`lbtout.xyz` or `lbtout.xy`) and a single set of solution files (`lbtout*.q`) for visualization of the entire system.

If `c++` is the command for the available C++ compiler, the executable `plot3d.exe` can be produced by typing

- `c++ -o plot3d.exe lbeplot3dgather.cpp`

and either run at the command line (`plot3d.exe` or `./plot3d.exe`) or via the GUI under **Gather LBE Data**.

All `lbout*.xyz` and `lbout*.q` files should be copied to the directory including the executable (if necessary) before running, as well as the `lbout.info` file to give information on the sizes of integers and floating point numbers. No user input is required, although the utility will stop with an error message if no `lbout.info` file is available. No other error messages are produced, so care should be taken to ensure no solution files are missing. Since the data is copied into the combined structure and solution files, the original output files can be deleted after this utility is run.

lbevtkgather

`lbevtkgather` is a utility written in C++ to gather Structured Grid XML-formatted VTK output files produced by the parallel version of DL_MESO_LBE (`lbout*.vts`) and produce a set of linking files (`lbtout*.pvts`) for

[1] If using the GUI and the utilties are to be compiled manually or in their source directories, copies of the executables are required in the directory from which the GUI is to be launched, e.g. `/WORK`.

visualization of the entire system.

If `c++` is the command for the available C++ compiler, the executable `vtk.exe` can be produced by typing

- `c++ -o vtk.exe lbevtkgather.cpp`

and either run at the command line or via the GUI under **Gather LBE Data**.

All `lbout*.vts` files should be copied to the directory including the executable (if necessary) before running, as well as the `lbout.info` and `lbout.ext` files to give information about the number of processors used for the simulation and the extents of each piece.

This utility can be run with a command line argument to give the scalar property required, e.g. for Windows and Unix/Linux computers respectively

- `vtk.exe 1`
- `./vtk.exe 1`

where 0 is used for fluid density, 1 for mass fraction, 2 for solute concentration and 3 for temperature. (If the GUI is used, this can be selected using the pulldown list in the **Gather LBE Data** panel.) If the argument is omitted, the utility will ask the user to enter the required property. No other user input is required, but error messages will be produced if either of the files `lbout.info` and `lbout.ext` are missing. No other error messages are produced, so care should be taken to ensure no VTK files for the pieces are missing, particularly since these files are required for plotting as the linking files do not include the data.

B.2 DL_MESO_DPD

molecule-generate

`molecule-generate` is a utility written in C++ to generate the input files required for modelling particles in DPD simulations that are bonded together, i.e. molecules. A random flight generation system is used to generate the coordinates of bonded beads – which can form branched molecule chains – a constant distance apart within a cube of a size specified by the user, which will be used by DL_MESO_DPD to insert the molecule into the system.

This utility can be compiled to produce the executable `molecule.exe` with the command

- `c++ -o molecule.exe molecule-generate.cpp`

if `c++` is the command for the available C++ compiler. This utility can be run from the command line or via the GUI in **Set DPD Molecules** (which runs the utility in a new command line/shell window).

When running the utility, the user will be asked for the number of molecules required. If a `CONTROL` file exists in the same directory as the executable, the number of species and their names will be read from it; otherwise the user will be asked to enter this information. For each molecule, the user is asked for the side length for the cube, bond length and number of molecule chains, followed by the number of particles for each chain. If the chain in question is not the first (primary) chain, the user will also be asked for a pre-existing bead number as the starting point for the chain and the neighbouring bead for determining bond angles. After this, the default species for the beads in the molecule will be requested: the user will then be asked enter the bead numbers for each of the other species (0 can be entered to finish specifying bead numbers).

`MOLECULE` and `FIELD` files will be generated. It should be noted that as generated by the utility:

- no bond, angle or dihedral interaction parameters are included in the `FIELD` file.

- the bond and bond angles are all of types 1.
- no bond dihedrals are included.

Some editing by the user may be required before the files can be used in DPD simulations. Section 11.3.1 gives the formatting of these files, and the example cases involving molecules (in the `DEMO/DPD` directory) should be examined by the user for guidance. If the data for additional molecule types are to be included, they should be appended to both the `MOLECULE` and `FIELD` files and the molecules *must* be in the same order in both files: up to eight types can be used by default in DL_MESO_DPD.

exportconfig

`exportconfig` is a utility written in Fortran90 to produce a configuration file in DL_POLY format (`CONFIG`) from DL_MESO_DPD restart files (`export*`), which can be used as a starting point for new simulations (including restarting simulations on different numbers of processes). Since a limited amount of data is included in restart files, the `CONTROL` file for the simulation is needed to provide some additional information.

The source code for this utility, `exportconfig.f90`, can be used for `export` or `export*` files created by both the serial and parallel versions of DL_MESO_DPD. If the available Fortran90 compiler is invoked by the command `f90`, the executable `config.exe` can be produced by typing

- `f90 -o config.exe exportconfig.f90`

and either run at the command line or by using the GUI in **Process DPD Data** after entering the number of processes used in the required field and selecting the required `CONFIG` file key in the pulldown list.

This utility can be run with two command line arguments, the first indicating the number of processes used to generate the restart data and the second denoting the `CONFIG` file key (`levcfg`: 0 = positions only, 1 = positions and velocities, 2 = positions, velocities and force), e.g. if 16 processes were used and the particle positions and velocities are required, either of the following commands can be used:

- `export.exe 16 1`
- `./export.exe 16 1`

If no command line argument is given, the utility will ask the user to type in the number of processes and the `CONFIG` file key.

exportimage

`exportimage` is a utility written in Fortran90 to produce a VTF format trajectory file (`export.vtf`) from DL_MESO_DPD restart files (`export*`) that can be visualized to give a snapshot of the last simulation timestep. Since a limited amount of data is included in restart files, the `CONTROL` file for the simulation is needed to provide some additional information.

The source code for this utility, `exportimage.f90`, can be used for `export` or `export*` files created by both the serial and parallel versions of DL_MESO_DPD. If the available Fortran90 compiler is invoked by the command `f90`, the executable `export.exe` can be produced by typing

- `f90 -o export.exe exportimage.f90`

and either run at the command line or by using the GUI in **Process DPD Data**.

This utility can be run with a command line argument indicating the number of processes used to generate the restart data, e.g. if 16 processes were used, either of the following commands can be used:

- `export.exe 16`
- `./export.exe 16`

If no command line argument is given, the utility will ask the user to type in the number of processes. (The number of processes can be specified in the GUI before running the utility.)

traject

The Fortran90 utility `traject` reads in `HISTORY*` output data files generated by DL_MESO_DPD and produces a VTF format trajectory file (`traject.vtf`) that can visualize the simulation after equilibration, such that snapshots at the recorded timesteps and animations can be produced.

Two versions of the utility are provided: `traject.f90` outputs every particle for all recorded timesteps, and `trajectselected.f90` allows the user to select the number of particles and the number of timesteps to output to the trajectory file. Both versions of the utility can be used with both `HISTORY` and `HISTORY*` files produced by the serial and parallel versions of DL_MESO_DPD respectively. If `f90` is the command for the available Fortran90 compiler, the executable `traject.exe` can be produced by typing

- `f90 -o traject.exe traject.f90`

and either run at the command line or via the GUI in **Process DPD Data**. A command line argument indicating the number of processes (and therefore the number of `HISTORY` or `HISTORY*` files) can be included in a similar way to the `exportimage` utility.

The alternative version can be compiled in the same way: by default the makefiles create executables named `trajects.exe`. This version does not take command line arguments and all information must be typed in by the user. After the number of processes is typed in, the utility displays the total number of particles and number of unbonded particles in the simulation, before asking for the particle number range and number of timesteps to be included in the trajectory file. This version of the utility cannot be invoked using the GUI.

local

`local` is a utility written in Fortran90 that can read in `HISTORY*` output data files generated by DL_MESO_DPD and produce series of VTK format files containing statistical properties – number of beads, density, compositions per particle and molecule types, pressure, temperature and mean velocity – in cuboidal subdivisions of the simulation volume for plotting and/or visualization.

The source code for this utility, `local.f90`, can read both `HISTORY` and `HISTORY*` files generated by the serial and parallel versions of DL_MESO_DPD respectively. If `f90` is the command for the available Fortran90 compiler, the executable `local.exe` can be produced by typing

- `f90 -o local.exe local.f90`

and either run at the command line or via the GUI in **Process DPD Data** after entering both the number of processes used in simulations and the number of divisions required in each dimension.

This utility can be run with four command line arguments: the first denoting the number of processes and the others giving the number of divisions required in x-, y- and z-directions respectively. If these arguments are omitted, the user will be asked to enter these values.

The following files are produced for all the specified time steps after equilibration:

- `scalars_*.vtk` with the numbers of beads, density, pressure and temperature for each cuboid cell

- `velocity_*.vtk` with the mean velocity of the beads within the cells
- `speccomp`i`_*`.vtk with the composition of species i in the cells (starting from species number 1)
- `molcomp`i`_*`.vtk with the composition of molecule type i in the cells (starting with number 0 for all unbonded particles)

The scalar properties (including compositions) may be considered to act across the entire volumes of the cells, while the velocities are representative for the cell centres.

Appendix C

DL_MESO Licence Agreement (Academic Purposes)

1. DEFINITIONS AND INTERPRETATION

1.1 In this Licence Agreement the following expressions have the meanings set opposite:

Academic Purposes	fundamental or basic research or academic teaching, including any fundamental research that is funded by any public or charitable body, but not any purpose that generates revenue (as opposed to grant income) for the Licensee or any third party. Any research that is wholly or partially sponsored by any profit making organisation or that is carried out for the benefit of any profit-making organisation is not an Academic Purpose;
a Derived Work	any modification of, or enhancement or improvement to, any of the DL_MESO Software and any software or other work developed or derived from any of the DL_MESO Software;
the DL_MESO Software	the release and version of the DL_MESO Software downloaded by the Licensee from the DL_MESO Website immediately after the Licensee agrees to the terms and conditions of this Licence Agreement;
the DL_MESO Website	the website with the URL http://www.ccp5.ac.uk/DL_MESO, and any website that from time to time replaces that website;
a Harmful Element	any virus, worm, time bomb, time lock, drop dead device, trap and access code or anything else that might disrupt, disable, harm or

impede the operation of any information system, or that might corrupt, damage, destroy or render inaccessible any software, data or file on, or that may allow any unauthorised person to gain access to, any information system or any software, data or file on it;

Intellectual Property — patents, trade marks, service marks, registered designs, copyrights, database rights, design rights, know-how, confidential information, applications for any of the above, and any similar right recognised from time to time in any jurisdiction, together with all rights of action in relation to the infringement of any of the above;

the Licence Period — the period beginning when the Licensee agrees to the terms and conditions of this Licence Agreement and downloads the DL_MESO Software from the DL_MESO Website and ending on the termination of this Licence Agreement under clause 5.2.

2. LICENCE

2.1 STFC grants the Licensee an indefinite, non-exclusive, non-transferable, royalty free licence to use, copy, modify, and enhance the DL_MESO Software during the Licence Period on the terms and conditions of this Licence Agreement provided that:

2.1.1 the Licensee may not distribute any of the DL_MESO Software or any Derived Work to any third party, or share their use with any third party (whether free of charge or otherwise), and the Licensee may not sub-license the use of any of the DL_MESO Software to any third party unless, in each case, that third party has complied with clause 2.3 below;

2.1.2 the Licensee may not use the DL_MESO Software on behalf of, or for the benefit of, any third party (including, without limitation, using it to provide bureau, outsourcing or application services or facilities management services) party unless that third party has complied with clause 2.3 below; and

2.1.3 the DL_MESO Software and any Derived Work may be used by the Licensee and its employees and registered students for Academic Purposes only.

2.2 If the Licensee wishes to use the DL_MESO Software or any Derived Work in any way except for Academic Purposes, or wishes to distribute or make the DL_MESO Software or any Derived Work

available to any third party for non-Academic Purposes, it must obtain a commercial licence from STFC. STFC may refuse to grant the Licensee a commercial licence. If STFC agrees to grant a commercial licence, that licence will be on such terms and conditions as STFC sees fit.

2.3 If the Licensee wishes to carry out any collaboration for Academic Purposes with any third party and that third party needs to use the DL_MESO Software in connection with that collaboration, or if the Licensee wishes to make the DL_MESO Software available online to any third party for Academic Purposes, the Licensee must direct that third party to the DL_MESO Website. That third party may use the DL_MESO Software and any Derived Work (whether obtained from STFC or from the Licensee) only if it has completed the registration process on the DL_MESO Website and agreed to the terms and conditions of the Licence Agreement for the use of the DL_MESO Software for Academic Purposes that then appear on the DL_MESO Website.

2.4 This Licence Agreement allows the Licensee to use only the release or version of the DL_MESO Software downloaded by the Licensee from the DL_MESO Website immediately after the Licensee agrees to the terms and conditions of this Licence Agreement; the Licensee must acquire a new licence for any future release or version of the DL_MESO Software that it wishes to use.

2.5 The Licensee will not tamper with, or remove, any copyright or other proprietary notice or any disclaimer that appears on or in any part of the DL_MESO Software, and will reproduce the same in all copies of any of the DL_MESO Software and in all Derived Works.

3. WARRANTIES AND LIABILITY

3.1 The DL_MESO Software is provided for Academic Purposes free of charge. Therefore STFC gives no warranty and makes no representation in relation to the DL_MESO Software or any assistance or advice that STFC may give in connection with the DL_MESO Software. The Licensee will indemnify STFC against any and all claims arising as a result of the Licensee having made any of the DL_MESO Software or any Derived Work available to any third party.

3.2 Before using any of the DL_MESO Software, the Licensee will check that the DL_MESO Software does not contain any Harmful Element. STFC does not warrant that the DL_MESO Software will run without interruption or be error free, or be free from any Harmful Element. STFC is not obliged to provide any support or error correction service, assistance or advice in relation to the DL_MESO Software, but the Licensee may access any error corrections and online assistance that STFC chooses to make

available on the DL_MESO Website from time to time. If STFC does provide that sort of service, assistance or advice, subject to clause 3.7, STFC will not be liable for any loss or damage suffered by the Licensee as a result.

3.3 STFC will not be liable to the Licensee to the extent that any loss or damage is caused: by the Licensee's failure to implement, or the Licensee's delay in implementing, any correction or advice in relation to the DL_MESO Software that STFC has made available on the DL_MESO Website; or by the Licensee's failure to acquire a licence of and to implement any new release or version of the DL_MESO Software that would have remedied or mitigated the effects of any error, defect, bug or deficiency in the DL_MESO Software.

3.4 The Licensee acknowledges that proper use of the DL_MESO Software and any Derived Work is dependent on the Licensee, its employees and students exercising proper skill and care in inputting data and interpreting the output provided by the DL_MESO Software or that Derived Work. STFC will not be liable for the consequences of decisions taken by the Licensee or any other person on the basis of that output. STFC does not accept any responsibility for any use which may be made by the Licensee of that output, nor for any reliance which may be placed on that output, nor for advice or information given in connection with that output.

3.5 Subject to clause 3.7, STFC's liability or any breach of this Licence Agreement, any negligence or arising in any other way out of the subject matter of this Licence Agreement or the use of the DL_MESO Software, will not extend to any incidental or consequential damages or losses, or any loss of profits, loss of revenue, loss of data, loss of contracts or opportunity, whether direct or indirect, even if the Licensee has advised STFC of the possibility of those losses arising or if they were or are within STFC's contemplation.

3.6 Subject to clause 3.7, the aggregate liability of STFC for any and all breaches of this Licence Agreement, any negligence or arising in any other way out of the subject matter of this Licence Agreement or the use of the DL_MESO Software will not exceed in total 5000.

3.7 Nothing in this Licence Agreement limits or excludes STFC's liability for death or personal injury caused by its negligence or for any fraud, or for any sort of liability that, by law, cannot be limited or excluded.

3.8 The express undertakings and given by STFC in this Licence Agreement and the terms of this Licence Agreement are in lieu of all warranties, conditions, terms, undertakings and obligations on the part of STFC, whether express or implied by statute,

common law, custom, trade usage, course of dealing or in any other way. All of these are excluded to the fullest extent permitted by law.

4. INTELLECTUAL PROPERTY RIGHTS AND ACKNOWLEDGEMENTS

4.1 Nothing in this Licence Agreement assigns or transfers any Intellectual Property Rights in any of the DL_MESO Software. Those rights are reserved to STFC.

4.2 The Licensee will ensure that, if any of its employees or students publishes any article or other material resulting from, or relating to, a project or work undertaken with the assistance of any part of the DL_MESO Software, that publication will contain the following acknowledgement:

"DL_MESO is a mesoscale simulation package written by R. Qin, W. Smith and M. A. Seaton and has been obtained from STFC's Daresbury Laboratory via the website http://www.ccp5.ac.uk/DL_MESO".

5. TERMINATION

5.1 This Licence Agreement will take effect and the Licence Period will start when the Licensee has agreed to the terms and conditions of this Licence Agreement and downloaded the DL_MESO Software from the DL_MESO Website.

5.2 This Licence Agreement will terminate immediately and automatically if:

5.2.1 the Licensee is in breach of this Licence Agreement; or

5.2.2 the Licensee becomes insolvent, or if an order is made or a resolution is passed for its winding up (except voluntarily for the purpose of solvent amalgamation or reconstruction), or if an administrator, administrative receiver or receiver is appointed over the whole or any part of its assets, or if it makes any arrangement with its creditors.

5.3 The Licensee's right to use the DL_MESO Software will cease immediately on the termination of this Licence Agreement, and the Licensee will destroy all copies of the DL_MESO Software that it or any of its employees or students then holds.

5.4 Clauses 1, 2.2, 3, 4, 5.3, 5.4, 5.5 and 6 will survive the expiry of the Licence Period and the termination of this Licence Agreement, and will continue indefinitely.

5.5 STFC may stop providing any assistance or advice in relation to, or any corrections, new releases or versions of the DL_MESO, and

may stop updating or publishing the DL_MESO Website at any time.

6. GENERAL

6.1 Headings: The headings in this Licence Agreement are for ease of reference only; they do not affect its construction or interpretation.

6.2 Assignment etc: The Licensee may not assign or transfer this Licence Agreement as a whole, or any of its rights or obligations under it, without first obtaining the written consent of STFC.

6.3 Illegal/unenforceable provisions: If the whole or any part of any provision of this Licence Agreement is void or unenforceable in any jurisdiction, the other provisions of this Licence Agreement, and the rest of the void or unenforceable provision, will continue in force in that jurisdiction, and the validity and enforceability of that provision in any other jurisdiction will not be affected.

6.4 Waiver of rights: If STFC fails to enforce, or delays in enforcing, an obligation of the Licensee, or fails to exercise, or delays in exercising, a right under this Licence Agreement, that failure or delay will not affect its right to enforce that obligation or constitute a waiver of that right. Any waiver by STFC of any provision of this Licence Agreement will not, unless expressly stated to the contrary, constitute a waiver of that provision on a future occasion.

6.5 Entire agreement: This Licence Agreement constitutes the entire agreement between the parties relating to its subject matter. The Licensee acknowledges that it has not entered into this Licence Agreement on the basis of any warranty, representation, statement, agreement or undertaking except those expressly set out in this Licence Agreement. The Licensee waives any claim for breach of, or any right to rescind this Licence Agreement in respect of, any representation which is not an express provision of this Licence Agreement. However, this clause does not exclude any liability which STFC may have to the Licensee (or any right which the Licensee may have to rescind this Licence Agreement) in respect of any fraudulent misrepresentation or fraudulent concealment before the signing of this Licence Agreement.

6.6 Amendments: No variation of, or amendment to, this Licence Agreement will be effective unless it is made in writing and signed by each party's representative.

6.7 Third parties: No one who is not a party to this Licence Agreement has any right to prevent the amendment of this Licence Agreement or its termination, and no one except a party to this Licence Agreement may enforce any benefit conferred by this

Licence Agreement, unless this Licence Agreement expressly provides otherwise.

6.8 Governing law: This Licence Agreement is governed by, and is to be construed in accordance with, English law. The English Courts will have exclusive jurisdiction to deal with any dispute which has arisen or may arise out of or in connection with this Licence Agreement, except that STFC may bring proceedings against the Licensee or for an injunction in any jurisdiction.

Bibliography

[1] Hans C. Andersen. Molecular dynamics simulations at constant pressure and/or temperature. *Journal of Chemical Physics*, 72(4):2384–2393, 1980.

[2] H. J. C. Berendsen, J. P. M. Postma, W. F. van Gunsteren, A. DiNola, and J. R. Haak. Molecular dynamics with coupling to an external bath. *Journal of Chemical Physics*, 81(8):3684–3690, 1984.

[3] Gerhard Besold, Ilpo Vattulainen, Mikko Karttunen, and James M. Polson. Towards better integrators for dissipative particle dynamics simulations. *Physical Review E*, 62(6):R7611–R7614, December 2000.

[4] P. L. Bhatnagar, E. R. Gross, and M. Krook. A model for collision processes in gases. I. Small amplitude processes in charged and neutral one-component systems. *Physical Review*, 94(3):511–525, May 1954.

[5] Shiyi Chen and Gary D. Doolen. Lattice Boltzmann method for fluid flows. *Annual Review of Fluid Dynamics*, 30(1):329–364, 1998.

[6] Paul J. Dellar. Bulk and shear viscosities in lattice Boltzmann equations. *Physical Review E*, 64(3):031203, August 2001.

[7] Dominique d'Humières, Irina Ginzburg, Manfred Krafczyk, Pierre Lallemand, and Li-Shi Luo. Multiple-relaxation-time lattice Boltzmann models in three dimensions. *Philosophical Transactions of the Royal Society of London. Series A: Mathematical, Physical and Engineering Sciences*, 360(1792):437–451, 2002.

[8] Burkhard Dünweg and Wolfgang Paul. Brownian dynamics simulations without Gaussian random numbers. *International Journal of Modern Physics C*, 2(3):817–827, 1991.

[9] U. Frisch, B. Hasslacher, and Y. Pomeau. Lattice-gas automata for the Navier-Stokes equation. *Physical Review Letters*, 56(14):1505–1508, April 1986.

[10] J. B. Gibson, K. Chen, and S. Chynoweth. The equilibrium of a velocity-Verlet type algorithm for DPD with finite time steps. *International Journal of Modern Physics C*, 10(1):241–261, February 1999.

[11] Minerva González-Melchor, Estela Mayoral, María Eugenia Velázquez, and José Alejandre. Electrostatic interactions in dissipative particle dynamics using the Ewald sums. *Journal of Chemical Physics*, 125(22):224107, 2006.

[12] R. D. Groot. Electrostatic interactions in dissipative particle dynamics—simulation of polyelectrolytes and anionic surfactants. *Journal of Chemical Physics*, 118(24):11265–11277, 2003.

[13] Robert D. Groot and Patrick B. Warren. Dissipative particle dynamics: Bridging the gap between atomistic and mesoscopic simulation. *Journal of Chemical Physics*, 107(11):4423–4435, 1997.

[14] Zhaoli Guo, Chuguang Zheng, and Baochang Shi. Discrete lattice effects on the forcing term in the lattice Boltzmann method. *Physical Review E: Statistical, Nonlinear and Soft Matter Physics*, 65(4):046308, April 2002.

[15] Cecil Hastings. *Approximations for digital computers*. Princeton University Press, Princeton, NJ, 1955.

[16] Xiaoyi He, Shiyi Chen, and Gary D. Doolen. A novel thermal model for the lattice Boltzmann method in incompressible limit. *Journal of Computational Physics*, 146(1):282–300, 1998.

[17] Xiaoyi He and Li-Shi Luo. Lattice Boltzmann model for the incompressible Navier-Stokes equation. *Journal of Statistical Physics*, 88(3–4):927–944, 1997.

[18] F. J. Higuera and J. Jiménez. Boltzmann approach to lattice gas simulations. *EPL (Europhysics Letters)*, 9(7):663–668, 1989.

[19] R. W. Hockney and J. W. Eastwood. *Computer simulation using particles*. McGraw-Hill International, 1981.

[20] Takaji Inamuro, Masato Yoshino, Hiroshi Inoue, Riki Mizuno, and Fumimaru Ogino. A lattice Boltzmann method for a binary miscible fluid mixture and its application to a heat-transfer problem. *Journal of Computational Physics*, 179(1):201–215, 2002.

[21] Takaji Inamuro, Masato Yoshino, and Fumimaru Ogino. A non-slip boundary condition for lattice Boltzmann simulations. *Physics of Fluids*, 7(12):2928–2930, 1995.

[22] Ask F. Jakobsen. Constant-pressure and constant-surface tension simulations in dissipative particle dynamics. *Journal of Chemical Physics*, 122(12):124901, 2005.

[23] J. E. Jones. On the determination of molecular fields. II. From the equation of state of a gas. *Proceedings of the Royal Society of London. Series A*, 106(738):463–477, October 1924.

[24] Simon Jury, Peter Bladon, Mike Cates, Sujata Krishna, Maarten Hagen, Noel Ruddock, and Patrick Warren. Simulation of amphiphilic mesophases using dissipative particle dynamics. *Physical Chemistry Chemical Physics*, 1(9):2051–2056, 1999.

[25] J. M. V. A. Koelman and P. J. Hoogerbrugge. Dynamic simulations of hard-sphere suspensions under steady shear. *EPL (Europhysics Letters)*, 21(3):363–368, 1993.

[26] Pierre Lallemand and Li-Shi Luo. Theory of the lattice Boltzmann method: Dispersion, dissipation, isotropy, Galilean invariance, and stability. *Physical Review E*, 61(6):6546–6562, June 2000.

[27] C. P. Lowe. An alternative approach to dissipative particle dynamics. *EPL (Europhysics Letters)*, 47(2):145–151, July 1999.

[28] John F. Marko and Eric D. Siggia. Stretching DNA. *Macromolecules*, 28(26):8759–8770, 1995.

[29] G. Marsaglia and T. A. Bray. A convenient method for generating normal variables. *SIAM Review*, 6(3):260–264, July 1964.

[30] George Marsaglia, Arif Zaman, and Wai Wan Tsang. Toward a universal random number generator. *Statistics and Probability Letters*, 9(1):35–39, January 1990.

[31] C. A. Marsh, G. Backx, and M. H. Ernst. Static and dynamic properties of dissipative particle dynamics. *Physical Review E*, 56(2):1676–1691, August 1997.

[32] Nicos S. Martys and Hudong Chen. Simulation of multicomponent fluids in complex three-dimensional geometries by the lattice Boltzmann method. *Physical Review E*, 53(1):743–750, January 1996.

[33] Makoto Matsumoto and Takuji Nishimura. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation*, 8(1):3–30, 1998.

[34] Keijo Mattila, Jari Hyväluoma, Tuomo Rossi, Mats Aspnäs, and Jan Westerholm. An efficient swap algorithm for the lattice Boltzmann method. *Computer Physics Communications*, 176(3):200–210, February 2007.

[35] Philip M. Morse. Diatomic molecules according to the wave mechanics. II. Vibrational levels. *Physical Review*, 34(1):57–64, July 1929.

[36] David R. Noble, Shiyi Chen, John G. Georgiadis, and Richard O. Buckius. A consistent hydrodynamic boundary condition for the lattice Boltzmann method. *Physics of Fluids*, 7(1):203–209, 1995.

[37] I. Pagonabarraga and D. Frenkel. Dissipative particle dynamics for interacting systems. *Journal of Chemical Physics*, 115(11):5015–5026, 2001.

[38] E. A. J. F. Peters. Elimination of time step effects in DPD. *EPL (Europhysics Letters)*, 66(3):311–317, May 2004.

[39] Kannan N. Premnath and John Abraham. Three-dimensional multi-relaxation time (MRT) lattice-Boltzmann models for multiphase flow. *Journal of Computational Physics*, 224(2):539–559, 2007.

[40] A. G. Schlijper, P. J. Hoogerbrugge, and C. W. Manke. Computer simulation of dilute polymer solutions with the dissipative particle dynamics method. *Journal of Rheology*, 39(3):567–579, 1995.

[41] Xiaowen Shan. Analysis and reduction of the spurious current in a class of multiphase lattice boltzmann models. *Physical Review E*, 73(4):047701, 2006.

[42] Xiaowen Shan. Pressure tensor calculation in a class of nonideal gas lattice Boltzmann models. *Physical Review E*, 77(6):066702, June 2008.

[43] Xiaowen Shan and Hudong Chen. Lattice Boltzmann model for simulating flows with multiple phases and components. *Physical Review E: Statistical Physics, Plasmas, Fluids, and related interdisciplinary topics*, 47(3):1815–1819, March 1993.

[44] Xiaowen Shan and Hudong Chen. Simulation of nonideal gases and liquid-gas phase transitions by the lattice Boltzmann equation. *Physical Review E: Statistical Physics, Plasmas, Fluids, and related interdisciplinary topics*, 49(4):2941–2948, April 1994.

[45] W. Smith. Molecular dynamics on hypercube parallel computers. *Computer Physics Communications*, 62(2-3):229–248, March 1991.

[46] W. Smith. A replicated data molecular dynamics strategy for the parallel Ewald sum. *Computer Physics Communications*, 67(3):392–406, January 1992.

[47] W. Smith. Calculating the pressure. *CCP5 Information Quarterly*, 39:14–20, October 1993.

[48] W. Smith, T. R. Forester, and I.T. Todorov. *The DL_POLY Classic user manual*. STFC, STFC Daresbury Laboratory, Daresbury, Warrington, Cheshire, WA4 4AD, United Kingdom, version 1.0 edition, December 2010.

[49] Simeon D. Stoyanov and Robert D. Groot. From molecular dynamics to hydrodynamics: A novel Galilean invariant thermostat. *Journal of Chemical Physics*, 122(11):114112, 2005.

[50] Sauro Succi. *The Lattice Boltzmann Equation for Fluid Dynamics and Beyond*. Clarendon Press, Oxford, 2001.

[51] Michael R. Swift, W. R. Osborn, and J. M. Yeomans. Lattice Boltzmann simulation of nonideal fluids. *Physical Review Letters*, 75(5):830–833, July 1995.

[52] I. T. Todorov and W. Smith. *The DL_POLY_4 user manual*. STFC, STFC Daresbury Laboratory, Daresbury, Warrington, Cheshire, WA4 4AD, United Kingdom, version 4.01.0 edition, October 2010.

[53] S. Y. Trofimov, E. L. F. Nies, and M. A. J. Michels. Thermodynamic consistency in dissipative particle dynamics simulations of strongly nonideal liquids and liquid mixtures. *Journal of Chemical Physics*, 117(20):9383–9394, 2002.

[54] Loup Verlet. Computer "experiments" on classical fluids. I. Thermodynamical properties of Lennard-Jones molecules. *Physical Review*, 159(1):98–103, July 1967.

[55] P. B. Warren. Vapor-liquid coexistence in many-body dissipative particle dynamics. *Physical Review E*, 68(6):066702, December 2003.

[56] P. B. Warren, P. Prinsen, and M. A. J. Michels. The physics of surfactant dissolution. *Philosophical Transactions of the Royal Society of London. Series A: Mathematical, Physical and Engineering Sciences*, 361(1805):665–676, 2003.

[57] John D. Weeks, David Chandler, and Hans C. Andersen. Role of repulsive forces in determining the equilibrium structure of simple liquids. *Journal of Chemical Physics*, 54(12):5237–5247, 1971.

[58] Satoru Yamamoto, Yutaka Maruyama, and Shi-aki Hyodo. Dissipative particle dynamics study of spontaneous vesicle formation of amphiphilic molecules. *Journal of Chemical Physics*, 116(13):5842–5849, 2002.

[59] M. Yoshino and T. Inamuro. Lattice Boltzmann simulations for flow and heat/mass transfer problems in a three-dimensional porous structure. *International Journal for Numerical Methods in Fluids*, 43(2):183–198, 2003.

[60] Qisu Zou and Xiaoyi He. On pressure and velocity boundary conditions for the lattice Boltzmann BGK model. *Physics of Fluids*, 9(6):1591–1598, June 1997.